PARMELEE PHARMACEUTICAL COMPANY et al. v. ZINK, doing business as SAFETY EQUIPMENT COMPANY

(CA 8)
128 USPQ 271

Decided Jan. 25, 1961
No. 16493
U.S. Court of Appeals Eighth Circuit

Headnotes

PATENTS

1. Courts of Appeals--Issues determined (§ 29.10)

While court of appeals is aware that Supreme Court has held that patent validity question has greater public importance than infringement question and that full inquiry into validity will usually be better practice and while court could entertain doubt as to validity of instant patent, court would encounter difficulty in passing upon validity inasmuch as district court made no findings as to validity and question is neither raised nor briefed on appeal; hence, court is disinclined formally to determine validity or to return case to district court for consideration thereof; court confines its comments to infringement.

2. Infringement--Tests of--Comparison with claim (§ 39.803)

In determining infringement, resort first must be had to words of claim; if accused device falls clearly within claim, infringement is made out, but that which is not literally within claim does not infringe.

3. Claims--"Comprising," "Consisting," etc. (§ 20.30)

In patent law, "comprising" is open-ended word and one of enlargement, not of restriction; in contrast, "consisting" is word of restriction and exclusion.

4. Infringement--Substitution of equivalents--In general (§ 39.751)

Doctrine of equivalents found initial expression in another day and under a statute somewhat different in its requirements as to claims from statute which now prevails; nevertheless, doctrine has a modern application.

5. Infringement--Substitution of equivalents--In general (§ 39.751)

Mere presence of equivalency is not in itself enough to warrant invocation of doctrine of equivalents nor does it necessarily equate with infringement; requirements of patent statutes must still, and initially, be met; thus, for example, doctrine cannot be used to expand confines of claim; cited law review article suggests that doctrine's proper application is where, because of formalized practice under statutes, claiming burden upon patentee is so inequitable as to merit extraordinary relief.

6. Infringement--Substitution of equivalents--Basic, improvement or paper patent (§ 39.753)

Doctrine of equivalents may be invoked not only where patent is a pioneer but also where secondary invention is involved, but range of equivalents depends upon and varies with degree of invention; where patent is a pioneer, patentee is allowed a wide range; where patent is narrow, or art is crowded, it is given a correspondingly narrow range; range of equivalents may be so narrow as to be virtually nonexistent.

7. Infringement--Substitution of equivalents--In general (§ 39.751)

Doctrine of equivalents may apply to chemical as well as to mechanical patents.

8. Construction of specification and claims--By Patent Office proceedings--In general (§ 22.151)

Infringement--Substitution of equivalents--In general (§ 39.751)

Doctrine of equivalents cannot be invoked to regain what was disclaimed in Patent Office proceedings.

9. Courts of Appeals--Weight given findings of District Court--Validity and infringement (§ 29.359)

Infringement--Substitution of equivalents--In general (§ 39.751)

Finding of equivalence is finding of fact; trial court's finding on the issue is not disturbed on appeal unless clearly erroneous.

10. Infringement--Substitution of equivalents--In general (§ 39.751)

Equivalence is not prisoner of a formula and is not an absolute; things equal to the same thing in this area may not necessarily be equal to each other.

11. Infringement--Substitution of equivalents--In general (§ 39.751)

If there had been any prior question about equivalency doctrine's current vitality, that question was resolved by Graver v. Linde, 339 U.S. 605, 85 USPQ 328, in favor of its continuing life and its applicability in proper situations.

Particular patents--Sodium Chloride

2,478,182, Consolazio, Sodium Chloride Tablet, not infringed.

Case History and Disposition:

Page 272

Appeal from District Court for Western District of Missouri, Ridge, J.; 126 USPQ 467 .

Action by Parmelee Pharmaceutical Company and William V. Consolazio against Lawrence C. Zink, doing business as Zink Safety Equipment Company, for patent infringement. From judgment for defendant, plaintiffs appeal. Affirmed.

Attorneys:
EDWARD A. HAIGHT, Chicago, Ill., for appellants.

JAMES C. WOOD, Chicago, Ill., for appellee.

Judge:
Before VAN OOSTERHOUT and BLACKMUN, Circuit Judges, and REGISTER, District Judge.

Opinion Text

Opinion By:
BLACKMUN, Circuit Judge.

This patent infringement action is one instituted by William V. Consolazio, patentee and owner, and Parmelee Pharmaceutical Company, exclusive licensee, against Lawrence C. Zink, d/b/a Zink Safety Equipment Company, seller of the accused product. Injunctive relief and damages are sought. The action is defended by Zink's manufacturer, Standard Safety Equipment Company; the defenses alleged are non-infringement, invalidity, and file wrapper estoppel. The District Court held that the patent was not infringed by the accused product and dismissed the complaint.

United States Letters Patent No. 2,478,182, issued to Consolazio August 9, 1949, is involved. It bears the title "Sodium Chloride Tablet". It matured from an application filed by Consolazio on January 16, 1945, and prosecuted by the Department of the Navy, [1] and it emerged from the Patent Office in the form of a single allowed claim reading as follows:

> "An internally reinforced sodium chloride tablet comprising compressed granules of sodium chloride; and an internally disposed cellular stroma of a thin, permeable, dialyzing film of a material selected from the group consisting of cellulose acetate and cellulose nitrate, the cells of said stroma containing said granules of sodium chloride, whereby the sodium chloride is rendered slowly available when the tablet reaches the gastro-intestinal tract, the solution time of the sodium chloride in said tablet in the gastro-intestinal fluids being from 60 to 80 minutes for a ten grain tablet."

The accompanying specifications refer to the use of salt tablets to combat the ill effects of heat and excessive sweating; the known incidence of epigastric discomfort, nausea and vomiting following their ingestion; the resort, as a consequence,

Page 273

by some steel mills to the alternative use of salted drinking water; the objections thereto; an earlier theory that the adverse symptons were due to irritation of the gastric mucosa by prolonged contact with salt; consequent attempts to develop a tablet which would disrupt and go into solution almost immediately and thus avoid prolonged contact; the lack of sucess of tablets of that kind; and the patentee's contrary concept of *retarding* the rate of absorption of salt with resulting elimination of discomfort. They also contain the usual general statement as to the invention. [2]

The patentee's tablet was commercially successful. The accused tablet, which appeared later and after advice of counsel that it did not infringe, conforms in general construction to the patentee's tablet and accomplishes the same result. The film employed in the defendant's tablet however, is *not* of "the group consisting of cellulose acetate and cellulose nitrate". It is, instead, "shellac". [3]

In its memorandum, 126 USPQ 467, supporting its dismissal of the complaint the District Court noted the application's long tenture (over 4 years) in the Patent Office, its original title "Tablets and Methods of Preparing Same", and the file wrapper history with its successive rejection of claims; characterized the art as crowded; and concluded that the patent was awarded only after the one claim had been limited to "a material selected from the group consisting of cellulose acetate and cellulose nitrate" and that, this being the case, the patent has no range of equivalents. The court thus entered its judgment of non-infringement without reaching and deciding what it called "the interesting issue of patentability of the plaintiffs' instant patent claim".

[1] We are aware that the Supreme Court has said that "of the two questions (validity and infringement), validity has the greater public importance" and that full inquiry into the question of validity "will usually be the better practice". Sinclair Co. v. Interchemical Corp., 325 U.S. 327, 330, 65 USPQ 297, 299. While we recognize the reason for this, viz, "that invalid claims shall not remain in terrorem of the art", Royal Typewriter Co. v. Remington Rand, 2 Cir., 168 F.2d 691, 77 USPQ 517, cert. den. 335 U.S. 825, 79 USPQ 454, and while we could entertain some doubt as to the validity of this patent, we would encounter difficulty, as did the court in the Royal Typewriter case, in passing upon the issue of validity on the record before us. There are no findings as to validity and the question is neither raised nor briefed on this appeal. As a consequence, we are disinclined formally to determine validity without the benefit of argument and the District Court's exercise of its judgment, or to return the case for consideration of that question. [4] We therefore confine our comments here to the problem of infringement. The plaintiffs argue in their main brief that the patented and the accused tablets are equivalent; that the trial court recognized this; that specific as well as generic patents are entitled to a reasonable range of equivalents; that the court's conclusion that the patent has no range at all is therefore error; that a patent claim is to be construed in the light of the substance of the advance contributed by the invention; that Consolazio's approach of controlled release was radically different from that of the past art; that he reduced this theory to practice with a stroma of thin permeable dialyzing film; that the choice of the material forming the film was only a matter of preference; that the invention did not lie in the choice of film material; that it was essentially a mechanical rather than a chemical improvement;

that the patent's own disclosure is in broad terms and does not teach that cellulose nitrate and cellulose acetate are the only effective film-forming materials; that the claim was finally allowed because it was more specific as to the physical properties of the structures than the earlier claims; that throughout the prosecution of the patent Consolazio urged the novelty of his tablet in physical structure, operation and results; that the patent is not entitled to a range of equivalents which would embrace every material which might be used to form a thin permeable dialyzing film but only those which are cellulose nitrate, cellulose acetate, or factually equivalent thereto; that there was no disclaimer by the applicant as to choice of film-forming material; and that the district court gave undue and misdirected emphasis to the word "consisting" as used in the allowed claim.

The defendant's response is that the patent in suit is not a pioneer but is one which emerged in a crowded art; that the prior art discloses the presence of film-forming materials not only as coating for the entire tablet (Davenport No. 617,956; Donard No. 874,310; Miller No. 2,011,587, and Kuever No. 2,373,763), but as coating for individual particles which make up the tablet (Davenport, supra; Donard, supra; and Andersen No. 2,410,417); that the claim of the patent necessarily measures its invention; that the patentee narrowed his claim in the Patent Office in order to gain its allowance and cannot now, through the doctrine of equivalents, recapture what was disclaimed; that he is subject to file wrapper estoppel; and that in any event the use of the word "consisting" in the claim and its reference to cellulose acetate and cellulose nitrate is restrictive and operates to exclude all substances not belonging to that group.

The plaintiffs' answer to these positive arguments of the defendant is that the invention was a radically new approach to the problem; that the prior art revealed only thick impermeable, nondialyzing coating placed on either tablets or their granules in order to achieve different results than those sought here; that whatever crowded character there was in the prior art did not relate to the substance of this invention; that the accused tablet does not eliminate any element of the patented combination but merely substitutes a different substance, equivalent in purpose and result, for cellulose acetate or cellulose nitrate; that this is a case for the application of the doctrine of equivalents; that the recital of specific materials in the claim was merely to satisfy formal requirements of the Patent Office; that the earlier rejected claims were just as specific regarding the identity of the impregnating material and hence its mere mention could not have been the trigger which released the patent; that the novelty was in the physical structure and operation rather than in identity of the film; and that the inventive combination of the claim in suit is introduced by the word "comprising" and not by the word "consisting", and hence any rule of restriction applying to the use of the latter word is not applicable here.

We have spelled out these respective contentions because they clearly reveal the points of conflict between the litigants. The plaintiffs' case rests on the proposition--admittedly unavoidable so far as they are concerned--that the substance of the invention and the patent must be the claimed new approach to the problem of salt intake, namely, the commencement of salt release upon ingestion and the continuance of salt release at a controlled rate through the use of a thin permeable film enveloping the salt granules and subject to dialytic action. The defendant does not deny that the patent involves a film with characteristics of this kind. If then Consolazio's patent is valid and applies to all thin permeable dialyzing films, or at least to those which are factually equivalent to cellulose nitrate and cellulose acetate, it could follow that the accused tablet, with its substitution of "shellac", does not eliminate a combinational element and does infringe. The case, therefore, pivots on the determination of just what the patent covers and comes down to the following:

> Does the patent's sole allowed claim by its very language confine its reach to films of the cellulose acetate and cellulose nitrate group?
>
> If so, is this a proper case for the application of the doctrine of equivalents and does that doctrine then afford this patent protection against the accused tablet?

We discuss these in order.

[2][3] 1. *The reach of the patent.* There are certain well established general rules which perhaps should be mentioned for background: (a) In determining whether an accused product infringes a patent "resort must be had in the first instance to the words of the claim". Graver Mfg. Co. v. Linde Co., 339 U.S. 605, 607, 85 USPQ 328, 328; General Bronze Corp. v. Cupples Products Corp., 8 Cir., 189 F.2d 154, 158, 89 USPQ 355, 355; Willis v. Town, 8 Cir., 182 F.2d 892, 893, 86 USPQ 138, 139. (b) If the "accused matter falls clearly within the claim, infringement is made out and that is the end of it". Graver Mfg. Co. v. Linde Co.,

supra, p. 607, 85 USPQ at 330. But, as this court has said, "that which is not literally within the claims does not infringe". Street v. Apel 8 Cir., 239 F.2d 581, 589, 112 USPQ 76, 80-81. (c) The word "comprising" in the patent law is an open-ended word and one of enlargement and not of restriction. "Claim 17 includes the expression 'loose granules of a natural material of the group comprising wood and grain'. The word 'comprising' does not exclude other materials besides wood and grains". Ex parte Dotter, 12 USPQ 382, 383-4. (d) In contrast, the word "consisting" is one of restriction and exclusion. " * * * (T)he words 'consists of' in claim 3 limit the gases of the tube, so far as claim 3 is concerned, to argon and mercury vapor". Claude Neon Lights v. Rainbow Light, 2 Cir., 90 F.2d 959, 961, 34 USPQ 140, 140; Ex parte Dotter, supra; Ex parte Jones and Swezey, 66 USPQ 487, 488-9. [5]

The application of these rules to the plaintiffs' tablet and to the accused tablet is revealing. There is no question that the latter is, to use the words of the Consolazio claim, "an internally reinforced sodium chloride tablet"; that it has "compressed granules of sodium chloride"; that it has "an internally disposed cellular stroma of a thin, permeable, dialyzing film"; that the cells of the stroma contain the granules of sodium chloride; that "the sodium chloride is rendered slowly available when the tablet reaches the gastro-intestinal tract"; and that the "solution time of the sodium chloride in said tablet in the gastro-intestinal fluids" is within the time range stated in the patent. [6] On the other hand, there is no argument either that the defendant's film of "shellac" is a material which is *not* from the cellulose acetate and cellulose nitrate group. There is thus not complete identity between the two tablets. Therefore, by this resort, as directed by Graver, to the words of the claim one cannot conclude that the "accused matter falls clearly within the claim" or that infringement automatically follows. [7] The plaintiffs can prevail, therefore, on the infringement issue only if this is a proper case for the application of the doctrine of equivalents.

2. *Equivalency.* The plaintiffs, strongly urge that the combination of elements in the claim and in the accused tablet are, in any event, "factually identical" although "specifically different in chemical identity" (and each with no chemical reaction between the salt and the film) and that the doctrine of equivalents works to their benefit here.

[4] The doctrine is not new to the patent law. It found initial expression in another day [8] and under a statute somewhat different in its requirements as to claims [9] from the one which now prevails. [10] Nevertheless, its age and the nature of the former statute do not deny the doctrine a modern application. Graver Mfg. Co. v. Linde Co., supra, 339 U.S. 605, 85 USPQ 328 .

An equivalent was defined long ago.

> "* * * (T)he substantial equivalent of a thing, in the sense of the patent law, is the same as the thing itself; so that if two devices do the same work
>
> Page 276
>
> in substantially the same way, and accomplish substantially the same result, they are the same even though they differ in name, form, or shape." Machine Co. v. Murphy, 1877, 97 U.S. 120, 125.

The same definition is used today. Graver, supra, p. 608 of 339 U.S., 85 USPQ at 330; Willis v. Town, 8 Cir., supra, p. 893 of 182 F.2d, 86 USPQ at 139. The reason behind the doctrine itself is apparent:

> "But courts have also recognized that to permit imitation of a patented invention which does not copy every literal detail would be to convert the protection of the patent grant into a hollow and useless thing. Such a limitation would leave room for--indeed encourage--the unscrupulous copyist to make unimportant and insubstantial changes and substitutions in the patent which, though adding nothing, would be enough to take the copied matter outside the claim, and hence outside the reach of law. * * *
>
> "The doctrine of equivalents evolved in response to this experience. The essence of the doctrine is that one may not practice a fraud on a patent." Graver, supra, pp. 607-8 of 339 U.S., 85 USPQ at 330.

[5] But the mere presence of equivalency is not in itself enough to warrant invocation of the doctrine nor does it necessarily equate with infringement. "The Doctrine of Equivalents Revalued", 1951, 19 George Washington Law Review, 491, 494, 503. The requirements of the patent statutes must still, and initially, be met; thus, for example, the doctrine cannot be used to expand the confines of a claim. James P. Marsh Corp. v. United States Gauge Co., 7 Cir., 129 F.2d 161, 165-6, 53 USPQ 653, 658. The law review note just cited suggests, pp. 492 and 504, that the doctine's proper application is where, because of formalized practice under present statutes, the claiming "burden upon the patentee is so inequitable as to merit some form of extraordinary relief".

[6][7] The doctrine may be invoked not only when the patent is a pioneer but also where a secondary invention is involved. Graver, p. 608 of 339 U.S., 85 USPQ at 330. It may apply to chemical as well as to mechanical patents. And

> "What constitutes equivalency must be determined against the context of the patent, the prior art, and the particular circumstances of the case. Equivalence, in the patent law, is not the prisoner of a formula and is not an absolute to be considered in a vacuum. It does not require complete identity for every purpose, and in every respect. In determining equivalents, things equal to the same thing may not be equal to each other, and, by the same token, things for most purposes different may sometimes be equivalents. Consideration must be given to the purpose for which an ingredient is used in a patent, the qualities it has when combined with the other ingredients, and the function which it is intended to perform. An important factor is whether persons reasonably skilled in the art would have known of the interchangeability of an ingredient not contained in the patent with one that was." Graver, supra, p. 609 of 339 U.S., 85 USPQ at 330-331.

In the same case the Supreme Court speaks in terms of degree when it says, p. 610, 85 USPQ at 331:

> "The question which thus emerges is whether the substitution * * * in view of the technology and the prior art, is a change of such substance as to make the doctrine of equivalents inapplicable; or conversely, whether under the circumstances the change was so insubstantial that the trial court's invocation of the doctrine of equivalents was justified."

Compare Sanitary Refrig'r Co. v. Winters, 280 U.S. 30, 42, 3 USPQ 40, 44.

[8][9]If the doctrine is applicable, then the range of equivalents depends upon and varies with the degree of invention. Paper Bag Patent Case, 210 U.S. 405, 415; Miller v. Eagle Manufacturing Co. 151 U.S. 186, 207. Thus, where the patent is a pioneer the patentee is allowed a wide range. Shakespeare Co. v. Perrine Mfg. Co. 8 Cir., 91 F.2d 199, 202, 34 USPQ 172, 175. But where the patent is narrow, or the art is crowded, it is given only a correspondingly narrow range. Electrol, Inc. v. Merrell & Co., 8 Cir., 39 F.2d 873, 878, 4 USPQ 497, 497; Ronson Patents Corp. v. Sparklets Devices, 8 Cir., 202 F.2d 87, 93, 96 USPQ 200, 200;R. H. Buhrke Co. v. Brauer Bros. Mfg. Co., 8 Cir., 33 F.2d 838, 839, 2 USPQ 104, 105. We have said on at least one occasion that the range of equivalents available to a patentee may "be so narrow as to be virtually nonexistent". Steffan v. Len A. Maune Company, 8 Cir., 234 F.2d 750, 753, 110 USPQ [8 7 9 , . And of course, the doctrine of equivalents cannot be invoked to regain what has been disclaimed in the Patent Office proceedings. Exhibit Supply Co. v. Ace Corp., 315 U.S. 126, 137, 52 USPQ 275, 280. Finally, a finding of equivalence is a finding of fact and a trial court's decision on the issue is

not to be disturbed unless clearly erroneous. Graver, supra, pp. 609-10 of 339 U.S., 85 USPQ at 331; Rule 52(a), F.R. Cv. P.

Judge Learned Hand has summarized well when he said in the Royal Typewriter case, supra, pp. 692-4 of 168 F.2d, 77 USPQ at 518-519:

> "* * * In these respects a patent is like any other legal instrument; but it is peculiar in this, that after all aids to interpretation have been exhausted, and the scope of the claims has been enlarged as far as the words can be stretched, on proper occasions courts make them cover more than their meaning will bear. If they applied the law with inexorable rigidity, they would never do this, but would remit the patentee to his remedy of reissue, and that is exactly what they frequently do. Not always, however, for at times they resort to the 'doctrine of equivalents' to temper unsparing logic and prevent an infringer from stealing the benefit of the invention. No doubt, this is, strictly speaking, an anomaly; but it is one which courts have frankly faced and accepted almost from the beginning. All patents are entitled to its benefit to an extent, measured on the one hand by their contribution to the art, and on the other by the degree to which it is necessary to depart from the meaning to reach a just result.
>
> "* * * It is true that a boundary cannot be drawn with precision; and the draftsman of claims is always in something of a dilemma--the dilemma which has led to the very 'doctrine of equivalents' itself.
>
> "* * * It is always a question of degree, and courts have differed, and always will differ, as to the allowable latitude in a given instance.* * *"

[10] Applying these standards to the case before us leads inevitably, it seems to us, to the conclusion that the doctrine of equivalents is of no help to these plaintiffs. We may go so far as to assume that a film of the cellulose acetate and cellulose nitrate group and the defendant's film of "shellac" do, in their respective tablets, perform the same function of controlled retardation of salt absorption. This then comes close to, if not within, the accepted definition of equavalency noted above. But, as the Supreme Court observed in Graver, p. 609, 85 USPQ at 330-331, equivalence is not the prisoner of a formula and is not an absolute, and things equal to the same thing in this area may not necessarily be equal to each other. We feel that the plaintiffs seek too much for their patent and we regard the prior art disclosed by the record and the file wrapper as crowded--in spite of plaintiffs' protestations that the crowding does not reach the impregnation aspect and their thin, permeable, dialyzing film--and the Consolazio patent as not a pioneer. In view of the same prior art, we feel that, so far as the doctrine of equivalents is concerned, the defendant's use of shellac was such a change of substance as to make the doctrine inapplicable and that, in the alternative, even if the doctrine does apply, the proper range of equivalents for this patent is a narrow one and, to use the language of Steffan, supra, is "so narrow as to be virtually non existent" or, to express ourselves perhaps more accurately for our present facts, is of insufficient breadth to include a substance outside "the group consisting of cellulose acetate and celullose nitrate". Cf. Miller v. Eagle Manufacturing Co., supra.

It follows then that, whether or not the District Court in its memorandum stated the law with complete accuracy when it said that the patentee here had disclaimed a number of equivalents and that this patent had "no range of equivalents", its finding that the patent was not infringed by the accused tablet is one not clearly erroneous within Rule 52 and is not to be set aside. General Bronze Corp. v. Cupples Products Corp., 8 Cir., supra, p. 158 of 189 F.2d, 89 USPQ at 357-358.

[11] We should and do note plaintiff's great stress and particular reliance on the Graver case to which many references have been made above. That is indeed a landmark decision, [11]although one by a divided court, for if there had been any prior question about the equivalency doctrine's current vitality, that question is now clearly resolved in favor of its continuing life and its applicability

in proper situations. The case's facts, however, are significant and deserve emphasis. We feel, as did the Second Circuit in its recent case of International Latex Corp. v. Warner Brothers Co., 2 Cir., 276 F.2d 557, 564, 124 USPQ 479, 484, cert. den. 364 U.S. 816, 127 USPQ 555 , that the Graver facts disclose "an entirely new type" of invented product; that as a consequence, the facts there are "quite different from those here, and that they do not direct us "to a contrary conclusion".

While one might ordinarily surmise that there is little or no connection between salt tablets and women girdles, we nevertheless regard the Latex case as fascinatingly parallel to this one. Spanel, the applicant there, was interested in a uniformity of stresses and found that by using a liquid latex material for the girdle's walls considerably thicker then theretofore employed, he obtained a material having a uniform stretch in every direction. He claimed advantages consisting, among other things, of higher tensile strength, lighter weight, absence of seams, more evenly distributed constricting effect, a smooth outer surface so that exterior garments would not tend to "ride up" and a reduction in the number of size models required. His application encountered difficulties in the Patent Office. There were repeated rejections of claims on grounds of undue multiplicity and prior art. Finally, after almost 4 years, the patent issued with certain of its claims relating to slightly roughened interior surfaces. The girdles proved very successful commercially. Defendant's girdles, after advice of counsel that they did not infringe, then appeared. These accomplished the departure from smoothness in the interior surface not by abrading, as the patent taught, but by a flocking process. The Second Circuit agreed with the trial court that the patent was valid, although roughening methods and other details had been known, in that the patentee was the first to develop a seamless latex girdle having such features that it produced the desired shaping effect even though not tailored to the particular form. Thus there were concepts of uniformity of force and avoidance of tailoring. But the court went on to hold that the accused product did not come within the plaintiff's claims because of the flocking as distinguished from abrading. The patent owner strongly urged the doctrine of equivalents. It was noted that the two processes served the same objectives, that the results were indistinguishable and that the only question was whether flocking was to be treated as an equivalent under the circumstances there. The court concluded that "it would be incongruous to apply an equitable doctrine" which would "relieve those who have failed to express their complete meaning". Graver was recognized but, as noted, was distinguished. The tribulations of the girdles and those of the salt tablets, strange as it may seem, are similar and we regard that case as pertinent precedent.

This renders it unnecessary for us to consider the additional arguments upon the issue of file wrapper estoppel.

Affirmed.

Footnotes

Footnote 1. Consolazio evidently developed his claimed invention while he was an officer in the United States Naval Research on active duty. In accord with the provisions of 35 U.S.C.A. § 266 the Government is given a royalty-free license by a recital in the patent itself.

Footnote 2. "According to my invention, the time of solution of tablets is prolonged by impregnating the tablets or coating the active ingredient thereof with water-insoluble, non-toxic, permeable, membranous films which are readily eliminated from the system by excretion in the feces. * * *

"The impregnating or coating agents which have been found to be most effective are cellulose derivatives, and, in particular, cellulose acetate and cellulose nitrate. * * *

"The cellulose acetate or nitrate penetrates the tablet forming a honeycomb structure around the salt granules. The granules are contained in small cellular compartments so that, in dissolution, fluid dialyzes into the cellular compartments, and salt or other enclosed material dialyzes out into the surrounding fluid, resulting in slow availability of the substance desired.

"Also, when the cellular compartments become engorged with fluid, these sacs burst and liberate the desired substance. With respect to the salt tablet, this mechanism results in a solution time of about 60 to 80 minutes for a ten grain tablet. When the tablet is completely dissolved, the cellular stroma of the impregnating film remains and is eliminated from the system in this form."

Footnote 3. We purposely place the word *shellac* in quotation marks. While there was testimony on behalf of the plaintiff to the effect that the defendant's impregnant was shellac, there was no admission during the course of the trial that this was so and the report as to the accused film's composition, required by the court to be filed beforehand in camera, was never opened.

Footnote 4. See Govero v. Standard Oil Co., 8 Cir., 192 F.2d 962, 963-4; Publicity Building Realty Corporation v. Hannegan, 8 Cir., 139 F.2d 583, 587.

Footnote 5. Compare Hoskins Mfg. Co. v. General Electric Co., N.D. Ill. 212 F. 422, 428, affirmed, 7 Cir., 224 F. 464, and In re Bertsch, CCPA 132 F.2d 1014, 1019-20, 56 USPQ 379, 384.

Footnote 6. The evidence indicated that a tablet's dissolution time was a factor within the control of any qualified chemist.

Footnote 7. The claim's unusual use of the two contrasting words of art, "comprising" and "consisting", merits examination. The plaintiffs urge that the broader word "comprising" is here applicable to "the inventive combination of the claim in suit", that the word "consisting" appears in the claim to introduce the group and that it was added only to satisfy a formal requirement of the Patent Office. They also urge that the word "consisting" has two different uses in patents; that, on the one hand, it is employed to introduce a combination of elements (where the invention resides in the combination); and that it is also used, as here, to introduce an incidental recital of the group disclosed as operable for forming the stroma.

While the matter may not be free from doubt, we are inclined to feel that, as used in this claim, the term "comprising" relates only to the immediately ensuing phrase "compressed granules of sodium chloride". The semicolon following the quoted phrase otherwise occupies no intelligible place in the wording of the claim. It would follow that the broader word "comprises" does not carry over to and affect the definition of "the group". The word "consisting", with its narrower import, is then the verbal adjective defining the group. And on the authority of the cases above cited the group, as so defined, is cellulose acetate and cellulose nitrate and nothing more.

In any event, as stated above, we conclude that the language of the patent's sole claim reaches only to cellulose acetate and cellulose nitrate and that the defendant's tablet is outside its terms.

Footnote 8. Winans v. Denmead, 1853, 15 How. 329, 343. 4.

Footnote 9. The Patent Act of 1836, § 6, Ch. 357, 5 Stat. 117, required that the claim only "specify and point" out the invention.

Footnote 10. 35 U.S.C.A. § 112 (the Act of July 19, 1952, Ch. 950, § 1 66 Stat. 798) :

> " * * * The specifications shall conclude with one or more claims particularly pointing out and distinctly claiming the subject matter which the applicant regards as his invention. * * * "

Footnote 11. As is perhaps evidenced by the case's travels through the courts. Linde Air Products Co. v. Graver Tank & Mfg. Co., N.D. Ind. 1947, 86 F.Supp. 191, 75 USPQ 231 , affirmed in part and reversed in part, 7 Cir., 1948, 167 F.2d 531, 77 USPQ 207 , affirmed in part and reversed in part, Graver Mfg. Co. v. Linde Co., 1949, 336 U.S. 271, 80 USPQ 451 , rehearing granted limited to the questions of infringement of four composition claims and the applicability of the doctrine of equivalents, 337 U.S. 910; adherence to prior decision, 1950, 339 U.S. 605, 85 USPQ 328 . Motion to adjudge defendants in contempt granted, Union Carbide & Carbon Corp. v. Graver Tank & Mfg. Co., N.D. Ind. 1951, 106 F. Supp. 389, 93 USPQ 158 , reversed, 7 Cir., 1952, 196 F.2d 103, 93 USPQ 137 , cert. den. 343 U.S. 967, 93 USPQ 536 , Trial Court's final judgment on damage issues reversed, Union Carbide Corporation v. Graver Tank & Mfg. Co., 7 Cir., 1960, 282 F.2d 653, 127 USPQ 3 , cert. applied for, Dec. 27, 1960.

- End of Case -

ISSN 1526-8535

FULL TEXT OF CASES (USPQ FIRST SERIES)
Ex parte Morrell, 100 USPQ 317 (BdPatApp&Int 1954)

Ex parte Morrell

(BdPatApp&Int)
100 USPQ 317

Patent issued Feb. 23, 1954
Opinion dated Mar. 19, 1953
U.S. Patent and Trademark Office, Board of Patent Appeals and Interferences

Headnotes

PATENTS

1. Claims-Broad or narrow-Markush type-Chemical (§ 20.2053)

Claims-"Comprising," "Consisting," etc. (§ 20.30)

Markush group must be definite and complete as to its membership; group is indefinite, and claims are rejected, where group is defined as "comprising * * *"; however, claims are allowed if "consisting of" is substituted for "comprising."

Particular patents-Hydrocarbons

2,670,321, Morrell, Conversion of Hydrocarbons with an Active Silica Composited Catalyst Mixture, claims 2, 4 to 6, 10 to 13, and 17 to 19 of application allowed; claims 1, 3, 7, 8, 14, 15, 20, and 21 refused.

Case History and Disposition:

Appeal from Division 31.

Application for patent of Jacque C. Morrell, Serial No. 79,527, filed Mar. 3, 1949. From decision rejecting claims 1 to 8, 10 to 15, and 17 to 21, applicant appeals. Affirmed as to claims 1, 3, 7, 8, 14, 15, 20, and 21; reversed as to remaining claims.

Attorneys:
Jacque C. Morrell, Chevy Chase, Md., pro se.

Judge:
Before Geniesse and Duncombe, Examiners in Chief, and Gonsalves, Acting Examiner in Chief.

Opinion Text

Opinion By:
Duncombe, Examiner in Chief.

This is an appeal from the final rejection of claims 1 to 8, 10 to 15, 17, 18, 19, 20, and 21, which are all of the claims in the case. Claims 2 and 20 are illustrative and are as follows:

Page 318

2. A process for the conversion of higher boiling to lower boiling hydrocarbons of high octane value and suitable for motor fuel which comprises subjecting the said hydrocarbons to conversion conditions of temperature and space velocities in the presence of a composited catalyst comprising an intimate and integrated mixture of active silica and active titanium oxide and a dehydrogenating component of an oxide in the physical state of a gel selected from the group comprising the active oxides of chromium, molybdenum and vanadium, the said oxides consisting of more than 1% and less than 10% of the mixture.

20. A catalyst comprising an intimate and integrated mixture of an active silica and an active titanium oxide composited with an oxide in the physical state of a gel selected from the group comprising the active oxides of chromium, molybdenum and vanadium, said oxides of chromium, molybdenum and vanadium consisting of more than 1% and less than 10% of the mixture and substantially not less than 50% silica of the mixture.

The references relied on are:

Pongratz, 2,157,965, May 9, 1939,

Melaven et al., 2,258,787, Oct. 14, 1941,

Young, 2,344,911, Mar. 21, 1944,

Arveson, 2,372,165, Mar. 20, 1945,

Spence et al., 2,385,552, Sept. 25, 1945,

Bates, 2,395,836, Mar. 5, 1946,

Michael et al., 2,500,197, Mar. 14, 1950.

The appealed claims relate to the heat treatment of hydrocarbons in the presence of a catalyst to convert them into lower boiling hydrocarbons, or to "reform" the hydrocarbons to improve their octane rating. The particular catalyst which is employed in the process of the elected claims on appeal is an intimate and integrated mixture of active silica and active titanium oxide and an oxide in the physical state of a gel selected from the group consisting of the active oxides of chromium, molybdenum and vanadium. The appeal also includes two claims, 20 and 21, which are drawn to the catalyst composition per se.

Claims 20 and 21 stand rejected as unpatentable over Pongratz in view of Spence et al.

Claim 20 calls for an intimate and integrated mixture of an active silica and an active titanium oxide composited with an oxide in the physical state of a gel selected from the group consisting of the active oxides of chromium, molybdenum and vanadium. The said gel is present in an amount of more than 1% and less than 10% of the mixture, and the silica is present to the extent of not less than 50% of the mixture.

Pongratz discloses a catalyst composition formed by precipitating titanium and vanadium salts on pumice and heating to about 360°C. Spence et al. disclose the precipitation of cromium or vanadium compounds in the form of gels, followed by slow drying to produce vitreous particles of catalyst. The catalyst may be "extended" by mixture with an extender such as silica, silica gel, pumice or diatomaceous earth. The position of the examiner is that since Spence et al. teach that silica gel is the equivalent of pumice in a catalyst composition there would be no invention in substituting silica gel for the pumice of Pongratz's catalyst composition. We are in agreement with this position. Appellant contends that the proportions recited in claim 20 and which are not shown by Pongratz are most important but a study of the record does not convince us that these proportions are critical. The objection is also made by appellant that Pongratz does not disclose the catalyst components in gel form. However it appears to us from a study of the art before us that the use of catalysts in gel form is common practice as shown, for example, by Spence et al., and does not involved invention. We will sustain the rejection of claim 20 as unpatentable over Pongratz in view of Spence et al.

Claim 21 specifies that the silica and titanium oxide are present in gel form. As noted above, Spence et al. disclose catalysts containing either chromium or vanadium compounds in gel form, and we, accordingly, find nothing patentable in this feature. We affirm the rejection of claim 21 as unpatentable over Pongratz in view of Spence et al.

Claims 2, 4 to 6, 10 to 13, and 17 to 19 stand rejected as unpatentable over Michael et al. The examiner relies particularly on lines 38 to 51, column 2, page 1 of the reference wherein it is disclosed that a catalyst may be formed from silica as a gel or from both silica and alumina gels "together with one or more of the following metals: Zn, Sn, Ti, V, Cr, Mo, W, Fe, Fi, and Co." While it presumably might be possible to form an anticipation of appellant's catalyst by piecing together appropriate parts of this disclosure we do not consider that there is sufficient teaching in the reference that out of the many combinations of catalyst components possible under the disclosure, both titanium oxide and an oxide selected from the class consisting of the oxides of chromium, molybdenum and vanadium could

advantageously be associated with silica to form a catalyst for the reactions called for by the appealed claims. We, accordingly, will not sustain the rejection of claims 2, 4 to 6, 10 to 13, and 17 to 19 as unpatentable over Michael et al.

[1] We note, however, that the Markush group (Ex parte Markush, 1925 C.D. 126) in claims 2, 4 to 6, 10 to 13, and 17 to 19 is indefinite as to scope in the use of the terms "comprising" in the phrase "comprising the active oxides of chromium, molybdenum and vanadium", instead of the words "consisting of." A Markush group must be definite and complete as to its membership, Ex parte Dotter, 12 USPQ 382. Under Rule 196(b) we, accordingly, reject claims 2, 4 to 6, 10 to 13, and 17 to 19 as reciting a Markush grouping which is not considered proper for the reason that the group is indefinite as to scope in the use of the term "comprising" instead of "consisting of." We recommend, however, that if appellant amends claims 2, 4 to 6, 10 to 13, and 17 to 19 to change "comprising" to "consisting of" in the parts of the claims which recite the Markush grouping, that said claims be allowed.

Claim 3 differs from the claims considered in the above paragraph in that it omits reference to an oxide selected from the group consisting of oxides of the metals, chromium, molybdenum and vanadium, calling merely for a mixture of silica, titanium oxide and alumina. We are of the opinion that the rejection of this claim as unpatentable over Michael et al. is sound and we will affirm it.

Claims 1, 7, 8, 14, and 15 stand rejected as not reading on the elected species. We do not find a traversal of this rejection in appellant's brief, and we will, accordingly, affirm it.

The decision of the Primary Examiner finally rejecting claims 1 to 8, 10 to 15, 17, 18, 19, 20, and 21 is affirmed in respect to claims 1, 3, 7, 8, 14, 15, 20, and 21 and is reversed in respect to claims 2, 4 to 6, 10 to 13, and 17 to 19.

Under Rule 196(b) we have entered a new rejection of claims 2, 4 to 6, 10 to 13, and 17 to 19; coupled, however, with a recommendation under Rule 196(c) that this group of claims be allowed if amended in the way we have suggested.

- End of Case -

ISSN 1526-8535

Entry: Arithmetic Keywords: + - * / ^ %% %/% Arithmetic arith ...
... Keywords: sys.parent sys.call sys.calls sys.frame sys.frames sys.nframe sys.function sys.parents sys.on.exit programming **data Description**: **Functions** to access ...
cvs.r-project.org/cgi-bin/viewcvs.cgi/*checkout*/ R/doc/html/search/Attic/index.txt?rev=1.1 - 52k - Supplemental Result - Cached - Similar pages

Google ▶
Result Page: 1 2 **Next**

+"data description functions" Google Search Search within results

Dissatisfied with your search results? Help us improve.

Google Home - Advertise with Us - Business Solutions - Services & Tools - Jobs, Press, & Help

Google Advanced Search Preferences Language Tools Search Tips

+"data description functions" Google Search

Web Images Groups Directory News

Searched the web for **+"data description functions"**. Results **11** - **14** of about **28**. Search took **0.22** seconds.

Entry: abbreviate Keywords: abbreviate character Description: ...
... sys.parent sys.call sys.calls sys.frame sys.frames sys.nframe sys.function sys.parents sys.on.exit sys.status programming **data Description**: **Functions** to access ...
cvs.r-project.org/cgi-bin/viewcvs.cgi/*checkout*/ R/doc/html/search/Attic/index.txt?rev=1.3 - 71k - Supplemental Result - Cached - Similar pages
[More results from cvs.r-project.org]

<html> <head> </head><body><pre>Entry: AIC Aliases: AIC AIC. ...
... calls sys.frame sys.frames sys.nframe sys.function sys.parents sys.on.exit sys.status parent.frame Keywords: programming **data Description**: **Functions** to Access ...
spijker.geog.uu.nl/docserv/rstats/search/index.txt - 101k - Cached - Similar pages

[PDF] Markets, Basic Statistics, Date and Time
File Format: PDF/Adobe Acrobat - View as HTML
... arFit Fit Univariate AR Models to Time Series **Data Description Functions** to simulate artificial AR time series processes, to fit the parameters of univariate ...
www.itp.phys.ethz.ch/econophysics/R/rSeries.pdf - Similar pages

[PDF] Network Working Group Richard Winter, Jeffrey Hill, Warren Greiff
File Format: PDF/Adobe Acrobat - View as HTML
Page 1. Network Working Group Richard Winter, Jeffrey Hill, Warren Greiff RFC # 610 CCA NIC # 21352 December 15, 1973 Further Datalanguage ...
www.faqs.org/ftp/rfc/pdf/rfc610.txt.pdf - Similar pages

In order to show you the most relevant results, we have omitted some entries very similar to the 14 already displayed.
If you like, you can repeat the search with the omitted results included.

◀ Google
Result Page: **Previous** 1 2

+"data description functions" Google Search Search within results

Google Home - Advertise with Us - Business Solutions - Services & Tools - Jobs, Press, & Help

Reading Assignment

In the yellow textbook: Venables and Ripley (MASS2), Chapter 1, thru section 1.2. If you will be using Splus for Windows at home, then you may also want to read section 1.3. If you don't have the book, then the early pages of Ripley's S-guide may suffice.

Laboratory for Week 1: Introduction to Splus

Starting Splus in the Microlab

You need to connect to your hsph UNIX account using telnet. I'd like you to use the `Simpterm' client which is available in the microlab's default Windows Desktop (the icon name for this is `telnet'). Log on to your hsph account, and type the following commands at the hsph prompt:

```
hsph% setenv S_CLEDITOR emacs
hsph% Splus -e
```

Now you are working in the Splus interactive environment and you can type Splus commands at the `>' prompt. The Splus option `-e' enables you to to scroll through and edit previously submitted commands using control characters. (Since we haven't submitted any commands yet, we'll have to try out this editing feature a little later in this session.)

Organization of directories and objects in Splus

Splus uses objects that are stored in several different locations which are kept in a search list. Here's how to see which locations you have in your search list:

```
> search()
```

The directory in position `[1]` is called your `working directory', and it is the default place where Splus stores objects that you create. Let's see what objects are stored in your working directory:

```
> objects(1)
```

There probably isn't much there because we haven't created any objects yet! (Note: There are a few objects which Splus automatically creates during your session that it uses to keep track of your work, the names of these special objects internal to Splus usually start with a `.'.)

A couple other things which you should know about creating objects in your working directory:

- The objects which you create during a session are *permanently* stored in your working directory. That is, unless you explicitly remove them with the `rm()' function, they will remain in the working

directory after the session has ended, and you will be able to access them again next time you start Splus.

- Also, if you create an object that has the same name as another object already in your working directory, the previously existing object will be overwritten by the new one, without even warning you. (Be careful!)

Also on the search list are several libraries which belong to the Splus language itself. This one probably has lot's of stuff in it, as it is one of Splus's language libraries:

```
> search()[4]
> objects(4)
```

(Note: The square brackets are used to subset a vector. Since the `search()` function returns a character vector, we can extract just the 4th element by appending the `search()` call with `` `[4]' ``. We'll use this type of subsetting a lot more later on, for all types of objects.)

- You can also use the `` `attach()' `` function to add new directories or named objects to the search list, so that the objects contained within them are also accessible to you during your current Splus session. By default, `attach()`ed objects go into position `[2]`, and all the locations below it move down on the list. A little later, we'll be talking about objects called `` `data.frames' ``, and they too can be `attach()`ed, giving us a convenient shorthand way to retrieve data...
- Finally, Splus uses the search list to look for objects which it needs, searching the search-list from *first* to *last*. It uses the first object it finds when there are name clashes. (Well, this is almost true: sometimes Splus knows it's looking for an object which also has a specified *class*, such as ``function", and it will use the first function it finds with that name rather than a non-function object which is higher on the search list.)

Loading a SAS data set into Splus

We have a some data from the NHANES1 study that we will be using throughout this course for the exercises. The data is stored as a SAS dataset in the `~eeb271/datasets` directory. Here is a description of the variables:

Description of the NHANES I Data	
age	age at enrollment
sex	gender (men 0; women 1)
race	race (categorical)
locode	geographic cluster identifier
height	height in
bmi	body mass index
booze	average alcohol consumption (categorical)
bp1sys	systolic blood pressure
calcium	dietary calcium consumption
ser.calc	serum calcium
ser.chol	serum cholesterol
smokenow	current smoking status (nonsmoker 0; current smoker 1)
smokever	smoking history (never smoker 0; ever smoker 1)
smokenum	current number of cigarettes smoked each day
smokpkyr	lifetime pack-years of cigarette smoking

Description of Epidemiologic Followup Data	
d.age	age at death
d.year	year of death
d.total	all cause death (alive 0; dead 1)
d.cancer	cancer death (alive or not cancer 0; cancer death 1)
d.ihd	ischemic heart disease death (alive or not heart 0; heart death 1)

Now we'd like to start exploring the data in Splus, but first we need to learn a bit about how Splus stores data.

Data frames

Splus has a nice way of packaging data into objects called `data.frames'. Data frames are analagous to SAS data sets, in that they are a *collection of named columns of data* (like SAS variables), which all have the same length, and their rows are indexed by an identifier called `row.names' (like an observation number or ID variable in SAS). Different columns of data can have different data types (called *modes* in Splus), including numeric, factor, or ordered factor.

The `sas.get()` function

The `sas.get()` function converts a SAS dataset into a data.frame. (It works by constructing and

submitting a special SAS program which reads your SAS dataset, and then converts it into an annotated ASCII file, which is then converted into a data.frame via Splus commands.) Note that **you need to also have SAS running on your computer** in order to use the `sas.get()` function, and I think `sas.get()` only works under UNIX. Windows users have other alternatives for converting SAS data to Splus data.frames.

A few more things about `sas.get()`:

- `sas.get()` knows how to convert SAS missing values, i.e., `.', to the appropriate Splus missing value token, `NA', which stands for `Not available'.
- `sas.get()` converts the underscores in variable names to dots.

Let's Create an Splus `data.frame` from an SAS dataset and check the working directory to see if our new object is there:

```
> hanes1_sas.get("/usr1/users/biostat/eeb271/datasets","hanes1")
> objects(1)
```

Now let's examine the `hanes1` object:

```
> class(hanes1)
> dim(hanes1)
> names(hanes1)
```

How many observations does the data.frame have? How many variables?

Now we can examine just one named column in the hanes1 `data.frame`. The numeric variables were stored by default as double precision, which is unnecessary for integer variables such as age. Changing the `storage.mode` to ``integer" will reduce the amount of disk-space the object takes up. The storage mode may be altered by assigning a new string to the storage.mode of that variable:

```
> mode(hanes1$age)
> storage.mode(hanes1$age)
> storage.mode(hanes1$age)<-"integer"
```

Now, let's attach the data.frame to position 3, and then list the objects that it contains:

```
> attach(hanes1, pos=3)
> search()
> objects(3)
```

Note that we now have an object named `age' on our search path, so instead of having to use the longhand `hanes1$age', we can simply access that object via the shorthand `age'. Please note that position 3 (where hanes1 has been attached) is not your working directory: if you modify any variables using the shorthand notation, the new versions will be stored in position 1. I recommend that you only make modifications to the variables using the longhand notation.

Here's an illustration of something very confusing which can happen if you don't follow my recommendation:

```
> find(age,numeric=T) # this tells us where age is on the search list
> age<-age/10         # let's modify age
> find(age,numeric=T) # now see where age is
> detach(3)           # this deletes the 3rd element on the search list
```

```
> find(age,numeric=T) # now see where age is
> hanes1$age[1:10]
> age[1:10]
```

Note: Comments can be made in Splus code using the # symbol; everything to the right of the # is a comment.

To reiterate: when using attach(hanes1), Splus *has made a copy* of the hanes1 dataset to a new path on the search list. If you want to modify any of the variables in the hanes1 object permanently, you must reference them using the longhand format: `hanes1$age`.

Data description functions

Calculate summary statistics for age and describe the distribution of age in this cohort:

```
> mean(age)
> median(age)
> quantile(age)
> quantile(age, c(0.1,0.9))
```

The summary() function provides a more efficient method for summary statistics.

```
> summary(age)
> page(summary(hanes1))
```

Create a categorical variable for age quintiles. This call to the `cut()` function chooses five equally spaced intervals of age:

```
> age5_cut(age, 5)
> table(age5)
```

This version of the cut() function chooses cut points that are quintiles of age.

```
> age5_cut(age, quantile(age, c(0,.2,.4,.6,.8,1)), include.lowest=T)
> table(age5)
> class(age5)
> summary(age5)
```

The categories have equal numbers of subjects, but are not equally spaced on the age axis. The `include.lowest` argument prevents the lowest values from being discarded.

Handling missing values

Examine the correlation of age with bmi. The following command will not run because bmi contains 3 missing values, as you should have see when you used the summary function before.

```
> cor(age,bmi)
> summary(bmi)
```

We can generate *logical vector* to use as an index to indicate which observations of data have a complete set nonmissing values for a set of variables. This logical vector may be incremented as necessary to account for additional variables.

```
> OK_!is.na(age) & !is.na(bmi) & !is.na(ser.chol) & !is.na(bp1sys)
> table(OK)
```

```
> OK_OK & !is.na(race) & !is.na(booze) & !is.na(sex) & !is.na(smokever)
> table(OK)
```

Remember the correlation that we could not run in the last exercise because of missing values? Now we can subset the bmi and age vectors to only those values with ÔK==T. Note: Splus knows how to convert logical vectors to binary vectors, mapping `F -> 0` and `T -> 1`, so the following two commands are equivalent:

```
> cor(age[OK],bmi[OK])
> cor(age[OK==1],bmi[OK==1])
```

Basic plotting functions

First we need to set up a graphics device. Until we get Splus for Windows installed here, we are going to use a device which I set up that uses ghostview on Windows to display the graphs. Please do the following:

1. You need to make a folder in the C: drive on your Windows machine for downloading the graphics files.
2. Set the `dowload directory' in Simpterm to be that folder.
3. Change the transfer option in Simpterm so that it overwrites files with name clashes, (This will enable your graphics to be updated automatically in your ghostview screen.)
4. Open ghostview in Windows; it is available from the `Start' menu, under `programs', `gstools', `gsview'.
5. I've set up a function called `ps()' which sends your graphs (in postscript format) via kermit to the PC. You need to attach the library which has that function:

```
> attach("/usr1/users/biostat/eeb271/.Data")
```

Now, let's make a histogram for age:

```
> hist(age, nclass=5, plot=F)
> ps(hist(age, nclass=5))
```

The default name of the file created by `ps()' is `temp.ps', and you should open that file in the ghostview window. Your graph should appear in the ghostview window. You can use the ghostview menu to save your graph to a different filename, or print to the printer in the microlab.

Please note that if you are not in the microlab and are using one of the other graphics devices, you need to omit the `ps()` call, and just submit:

```
> hist(age, nclass=5)
```

Before you Leave

For those of you who are planning to do your homework on another computer, you will need to get the data onto that computer. Here's a way to save a portable version of the data which can be read into Splus on any platform.

First, here's how to dump the Splus dataset to an ASCII file. This ASCII dataset can now moved using ftp or floppy diskette to any platform including another Unix system or a PC.

```
data.dump("hanes1","hanes1.dmp")
```

The file you need transfer or put onto diskette is located in your current directory, and called ``mydata.dmp".

Now, to restore the Splus dataset using Splus on the other computer,

```
> data.restore("hanes1.dmp")
> objects(1)
```

And your data should have been converted back into an Splus data.frame object.

Homework for Week 1

Please note that you may need to review/repeat some of the exercises in lab 1 in order to get your terminal set up properly to display graphics.

1. Compute quantiles and boxplots for serum cholesterol:

   ```
   > median(ser.chol)
   > quantile(ser.chol)
   > quantile(ser.chol, c(0.1,0.9))
   > boxplot(ser.chol)
   > boxplot(log(ser.chol))
   ```

   What is the distribution of serum cholesterol? Does the distribution of log(ser.chol) look closer to the normal distribution?

2. Does serum cholesterol vary with age? Describe the apparent relationship of serum cholesterol with age as shown by a scatter plot.

   ```
   > plot(ser.chol, age)
   ```

3. Plot the distribution of systolic blood pressure for each category of booze. Booze is a numeric vector with 0 for the lowest level of alcohol consumption and 3 for the highest.

   ```
   > boxplot(split(bp1sys, booze), notch=T)
   > boxplot(split(bp1sys, booze), varwidth=T)
   ```

   Which plot is most informative regarding the relationship of these variables? Using this same strategy, now make a boxplot for serum cholesterol by age quintiles.

4. In the previous questions, you've looked at the variable serum cholesterol in several ways. Please synthesize your findings in a short 2-5 sentence paragraph that would be suitable for publication.

Next: Week 2 **Up:** eeb99 **Previous:** Contents **Contents**

cwager@hsph.harvard.edu
1999-03-23

Next: Contents **Contents**

EEB271c: Advanced Regression Techniques for Environmental Health

Computing Laboratory
Teaching Assistant: Carrie Wager
Spring 1999

Disclaimer and Acknowledgement: *The material contained on this website has been revised from material prepared by previous lecturers for the course, It is a collaboration of the work of many people, including Joel Schwartz, Louise Ryan, and Lucas Neas, as well as myself.*

You can also download this document as a single file (eeb99.ps.gz, eeb99.pdf).

        - Some UNIX commands
        - source code and output for Splus Example
        - Running Splus in BATCH mode

- Week 5
    - Overview for Week 5: Nonparametric smoothing example
        - Smoothing Functions
        - GAM functions
            - Model Creation and Development
            - Extracting Model Components
        - GAM objects
            - Model description
            - Model results
        - Summarizing model results
            - Summary Functions
            - Summary Object
        - Akaike Information Criteria
    - Laboratory for Week 5: Nonparametric smoothing example
        - Basic glm model for all cause mortality.
        - Defining the `aic()` function
        - Parametric smoothing for serum cholesterol
        - Nonparametric smoothing for serum cholesterol
        - Serum cholesterol and cause-specific mortality
    - Homework for Week 5
        - Effect of pre-clinical cancer on serum cholesterol

- Week 6
    - Preparation for Week 5: Advanced Generalized Additive Models in SPLUS
        - Stepwise selection of GAM models
        - Generalized Estimating Equations (GEE)
        - Random Effects in Linear Models
        - Linear Mixed Effect Object
    - ComputingLab for Week 6: Generalized additive model example
        - Bivariate smoothing
        - Using `step.gam` to find the best smoothing span
        - Linear Models for Blood Pressure
        - Generalized Estimating Equations for Blood Pressure
        - Random effects model for blood pressure
        - Generalized Linear Models for Mortality
        - Generalized Estimating Equation for Mortality

- Week 7
    - Preparation for Week 7: Time Series Analysis in Splus
        - Philadelphia mortality time series data set
        - Time series functions
    - ComputingLab for Week 7: Time series decomposition
        - Preliminaries
        - Making titles for GAM plots
        - Trend and season
        - Air pollution

EEB271c: Advanced Regression Techniques for Environmental Health

Computing Laboratory

Teaching Assistant: Carrie Wager

Spring 1999

Disclaimer and Acknowledgement: *The material contained on this website has been revised from material prepared by previous lecturers for the course, It is a collaboration of the work of many people, including Joel Schwartz, Louise Ryan, and Lucas Neas, as well as myself.*

Contents

1 Week 1

1.1 Preparation for Week 1

1.1.1 A brief history of Splus

The statistical language called *S* was developed by researchers at AT&T Bell labs (now Lucent Technologies) in the early 1980's for rapid prototyping of new statistical methodology and exploratory graphics techniques.
S used to be freely distributed by Bell Labs, but then in 1991 a company called Statistical Sciences bought the license to it, and began marketing versions 3.1 and 3.2 as enhanced products, with additional data-analytic and graphical capabilities.
Mathsoft bought Statistical Sciences in 1994, and is the current developer, marketer, and distributor of Splus, although researchers at Lucent Technologies still play a large role in developing Splus to accommodate the growing needs of the future. Also, many of the data modeling packages available for Splus are very recent developments contributed by statistical researchers all over the world, and are available for free downloading over the Internet.

1.1.2 Why use Splus rather than other statistical languages?

- S-PLUS is designed around an **object-oriented** framework. This means that all the data, the modelling and manipulation procedures (called *functions* in Splus), and the fitted models themselves are handled as objects. Objects have *classes* associated with them, and these classes identify them so that they can be recognized by specific *methods* for doing standard procedures (such as printing, graphing, or making reports) which have been designed for their class.

- Because all of the Splus source code is available to the user to view and potentially customize to their needs, this makes Splus very flexible as a data analysis environment. In combination with the object-oriented design, this also makes Splus useful for rapid prototype development of new statistical methods: newly developed methodologies are often available in Splus long before being incorporated into other languages such as SAS or Stata.

- Splus also has extensive, flexible, portable, and customizable graphics capabilities which appeal to users who are doing exploratory data analysis.

The Mathsoft site has a nice sales pitch[1] about Splus.
Also, if you are a SAS user, here is a comparison between SAS and Splus[2].

[1] http://www.mathsoft.com/splus/splsprod/spls5.html
[2] http://www.mathsoft.com/splus/splus_comparison.htm

Multiple operating platforms Mathsoft has recently released Splus version 5.0 for several platforms. This new release has a graphical user-interface on all platforms including UNIX. Most importantly, version 5 makes use of new technologies developed by researchers at Lucent Technologies which greatly improve the performance of Splus, especially in regard to handling very large (potentially gigabyte-sized) datasets. This capacity will be pertinent to us in EEB, since we'll be using the NHANES data, the full version of which has 4197 observations and 21 variables. In past years, having 20 students load this dataset simultaneously during the computer lab has caused us to crash the entire hsph UNIX system!

- **Splus version 3.4 UNIX** This is the main version we will be using for the EEB course, particularly for the workshops which are taught in the microlab. It has an interactive command-line interface.
- **Splus version 5 UNIX** This will be installed on hsph some time in the near future. We may be able to try it out for the course, although your lab instructor is not familiar with the graphical interface yet!
- **Splus version 4.5 Windows** This version is the current Windows release, and some students may be using it already. This is available for purchase at student/group rates. (See below1.1.2 for details.) It will also be installed on the microlab PC's sometime in the near future, so we may be able to try it out during the course.
- **Splus version 5 Linux** Linux is a free Unix-like OS available for PC's. Splus has just been made available on this platform for the first time. This may be available for purchase at student/group rates. (See below1.1.2 for details.)

The latest information on available Splus distributions can be found at MathSoft[3]'s website.

Buying your own copy of Splus software Mathsoft offers individual licenses to Splus verion 4.5 for Windows to students for $199 each. If we can get 10 or more students together for an order, this is discounted by 10%. Please put your name on the signup sheet by Thursday, February 4 if you would like to purchase Splus.

1.1.3 Getting Started

NOTE: *In this workshop, I will focus on the environment that we* currently *have available to us in the microlab computing facility. We will be using networked PC's running Windows, and the PC's do not yet have Splus version 5 for Windows installed on them, so we will connect (via telnet) to your hsph UNIX account, and we will be running Splus version 3.4 for UNIX from there.*
We are somewhat restricted in our ability to display remote graphics from the UNIX accounts to these Windows terminals, so, as the course progresses, we will be trying out a few different devices which the instructors have set up.

[3]http://www.mathsoft.com/Splus

Starting and Quitting an Splus Session in UNIX To start Splus, type 'Splus -e' at the hsph prompt:

```
hsph% Splus -e
Initializing login directory for new S-PLUS user.
S-PLUS : Copyright (c) 1988, 1996 MathSoft, Inc.
S : Copyright AT&T.
Version 3.4 Release 1 for Sun SPARC, SunOS 5.3 : 1996
Working data will be in /usr1/users/biostat/eeb271/.Data
>
```

Splus will give you a new prompt '>' at which you can type Splus commands interactively. If this is the first time you've ever started Splus, notice in the startup message that a directory was created for you in your account on the Unix filesystem. Splus will use it to store binary versions of all the objects you create; you can only view the contents of these objects from within Splus.
To **quit** out of Splus, type:

```
> q()
hsph%
```

...and you will be returned to the UNIX prompt.

Creating and viewing objects Let's create a few objects:

```
> X <- 1:10
> Y_rnorm(10)
> X
 [1]  1  2  3  4  5  6  7  8  9 10
> Y
 [1] -0.30744768  0.55918699  2.69330759  1.09345678  0.09873739 -0.91776485
 [7] -1.76152800  0.30370197 -0.52486689  1.46745534
```

The assignment operator can be specified either of 2 ways: the underscore, '_', is shorthand for lazy typists, while '<-' is easier to read. Here we have *assigned* the value of the *expression* on the r.h.s. to the *object* named on the l.h.s. Note that in order to display an object, all you have to do is type it's name at the prompt.
Now, let's make a function object and apply it to our data:

```
> myfun <- function(a,b)a/(a+b)
> myfun(X,Y)
 [1] 1.0208300 2.0241540 0.7660276 1.5284589 1.1036438 1.1546431 0.8857799
 [8] 1.1688848 0.9884700 0.9774965
> Xpct <- myfun(X,Y)
```

Here we've applied myfun to the 'X' and 'Y' vectors, and displayed the result, then assigned the result to the object 'Xpct'. We can see a list of the objects we have created with the 'objects()' function:

```
> objects()
[1] ".Last.value"  ".Random.seed" "X"            "Y"
[5] "myfun"        "result"
```

Getting Help Splus has an interactive help feature. In order to use it, you have to know the name of the command you want help for. The function which does least squares regression is called `lm()`; to find out more about it, type:

```
> help(lm)
```

or use the shorthand:

```
> ?lm
```

1.1.4 Setting up your graphics device

To display any of the graphs you make in Splus, you need to open a graphics device. Graphics device drivers are needed to translate pictures into something that can be rendered on your screen, or on a piece of paper, or some other media. The type of device you can use depends on what kind of computer you are sitting in front of.

- **Something that works for any platform**

  ```
  > printer(80,25)
  ```

  This starts up a graphics device which simply sends ASCII output to your screen. (You may need to use the 'show()' command to view a graph after creating it.) If all else fails, use this device; the output is crude, but it always works!

- **PC with serial (modem) connection to UNIX, using ghostview on PC to display**
  I've set up a function called 'ps()' which sends your graphs (in postscript format) via kermit to the PC. You should make a folder in the C: drive on your Windows machine for downloading the graphics files, and set the 'dowload directory' in Simpterm to be that folder. You need to open ghostview in Windows to view these graphics files. Ghostview is available from the 'Start' menu, under 'programs', 'gstools', 'gsview'.

  The default name of the file created by 'ps()' is 'temp.ps', and you should open that file. If you change the transfer option in Simpterm so that it overwrites files with name clashes, the file will be updated automatically in your ghostview screen each time you create a new graph.

- **PC with TCP/IP connection to UNIX, using vnc to start an x-session**
  We will talk about this option later if we need to use it.
- **PC with Splus Version 4.5 or Version 5 Splus for Windows**
  Type '`win.graph()`' at the Splus prompt. There is also a way to start up a graphics device using the menus.
- **Xterminal running Splus version 3.4 on UNIX**
  Type '`motif()`' at the Splus prompt.
- **Xterminal running Splus version 5 on UNIX**
  Type '`motif()`' at the Splus prompt. There's probably also a way to do this with the menus.

The command

```
> dev.off()
```

closes the current graphics device. (Note: you do not need to use '`dev.off()`' with '`ps()`', as it is done for you automatically.)

1.1.5 Ways to get a record of your session and output

- **Getting a history of your commands**

  ```
  > history()
  ```

  gives you a history of your last 10 commands. You can resubmit them by selecting the appropriate number from the menu.

- **'`sink()`ing' your output to a file**
  To redirect output to a file instead of your screen:

  ```
  > sink("myout.out")
  > options(echo=T)
  ```

  All of your output will now be sent to the file instead of the screen. Your commands will be sent to both the screen and the file because you have set the '`echo`' option. To return to the normal screen-output state, type:

  ```
  > options(echo=F)
  > sink()
  ```

1.1.6 Command line Editing

The command line editing feature especially useful when running Splus on a Unix computer. You may need to add the following line to your `.cshrc` file in your root directory (Ask the lab assistant for help with this if you don't know an editor.):

```
setenv S_CLEDITOR emacs
```

and then type the following at the hsph prompt:

```
source ~/.cshrc
```

Now you will be able to edit previously submitted commands using the following control sequences. (Note that these same control sequences work for editing commands submitted to the UNIX command prompt.)

`^P` recall previous line (Can be repeated to scroll through past lines.)

`^B` moves cursor nondestructively back one space

`^F` moves cursor nondestructively forward one space

`^K` erase from cursor to the end of the line

`^D` delete character at cursor

`^A` moves cursor to the beginning of the line

`^C` cancels command line editing and returns to a prompt

1.1.7 More functions

I'm just going to list the names of a few functions you may find useful in Splus. To find out more about them, use the '`help()`' command.

Data manipulation: `sas.get()`, `read.table()`

Descriptive statistics: `min()`, `max()`, `mean()`, `median()`, `quantile()`, `summary()`, `var()`, `cor()`

Plotting `stem()`, `hist()`, `boxplot()`, `plot()`, `plot.xy()`, `qqplot()`, `qqnorm()`

Mathematics: `exp()`, `log()`, `sqrt()`, `round()`

Statistical distributions: (Four types of functions do the PDF, CDF, quantiles, and random number generation, respectively)

F distribution: `df()`, `pf()`, `qf()`, `rf()`

Normal distribution: `dnorm()`, `pnorm()`, `qnorm()`, `rnorm()`

1.1.8 Reading Assignment

In the yellow textbook: Venables and Ripley (MASS2), Chapter 1, thru section 1.2. If you will be using Splus for Windows at home, then you may also want to read section 1.3. If you don't have the book, then the early pages of Ripley's S-guide[4] may suffice.

1.1.9 Course Website

This laboratory is also available online on the course website:

http://biosun1.harvard.edu/~eeb271

[4] http://lib.stat.cmu.edu/S/sguide.ps1

1.2 Laboratory for Week 1: Introduction to Splus

1.2.1 Starting Splus in the Microlab

You need to connect to your hsph UNIX account using telnet. I'd like you to use the 'Simpterm' client which is available in the microlab's default Windows Desktop (the icon name for this is 'telnet'). Log on to your hsph account, and type the following commands at the hsph prompt:

```
hsph% setenv S_CLEDITOR emacs
hsph% Splus -e
```

Now you are working in the Splus interactive environment and you can type Splus commands at the '>' prompt. The Splus option '-e' enables you to to scroll through and edit previously submitted commands using control characters. (Since we haven't submitted any commands yet, we'll have to try out this editing feature a little later in this session.)

1.2.2 Organization of directories and objects in Splus

Splus uses objects that are stored in several different locations which are kept in a search list. Here's how to see which locations you have in your search list:

```
> search()
```

The directory in position [1] is called your 'working directory', and it is the default place where Splus stores objects that you create. Let's see what objects are stored in your working directory:

```
> objects(1)
```

There probably isn't much there because we haven't created any objects yet! (Note: There are a few objects which Splus automatically creates during your session that it uses to keep track of your work, the names of these special objects internal to Splus usually start with a '.'.)
A couple other things which you should know about creating objects in your working directory:

- The objects which you create during a session are *permanently* stored in your working directory. That is, unless you explicitly remove them with the 'rm()' function, they will remain in the working directory after the session has ended, and you will be able to access them again next time you start Splus.
- Also, if you create an object that has the same name as another object already in your working directory, the previously existing object will be overwritten by the new one, without even warning you. (Be careful!)

Also on the search list are several libraries which belong to the Splus language itself. This one probably has lot's of stuff in it, as it is one of Splus's language libraries:

```
> search()[4]
> objects(4)
```

(Note: The square brackets are used to subset a vector. Since the `search()` function returns a character vector, we can extract just the 4th element by appending the `search()` call with '`[4]`'. We'll use this type of subsetting a lot more later on, for all types of objects.)

- You can also use the '`attach()`' function to add new directories or named objects to the search list, so that the objects contained within them are also accessible to you during your current Splus session. By default, `attach()`ed objects go into position `[2]`, and all the locations below it move down on the list. A little later, we'll be talking about objects called '`data.frames`', and they too can be `attach()`ed, giving us a convenient shorthand way to retrieve data...

- Finally, Splus uses the search list to look for objects which it needs, searching the search-list from *first* to *last*. It uses the first object it finds when there are name clashes. (Well, this is almost true: sometimes Splus knows it's looking for an object which also has a specified *class*, such as "function", and it will use the first function it finds with that name rather than a non-function object which is higher on the search list.)

1.2.3 Loading a SAS data set into Splus

We have a some data from the NHANES1 study that we will be using throughout this course for the exercises. The data is stored as a SAS dataset in the `~eeb271/datasets` directory. Here is a description of the variables:

Description of the NHANES I Data	
age	age at enrollment
sex	gender (men 0; women 1)
race	race (categorical)
locode	geographic cluster identifier
height	height in
bmi	body mass index
booze	average alcohol consumption (categorical)
bp1sys	systolic blood pressure
calcium	dietary calcium consumption
ser.calc	serum calcium
ser.chol	serum cholesterol
smokenow	current smoking status (nonsmoker 0; current smoker 1)
smokever	smoking history (never smoker 0; ever smoker 1)
smokenum	current number of cigarettes smoked each day
smokpkyr	lifetime pack-years of cigarette smoking

Description of Epidemiologic Followup Data	
d.age	age at death
d.year	year of death
d.total	all cause death (alive 0; dead 1)
d.cancer	cancer death (alive or not cancer 0; cancer death 1)
d.ihd	ischemic heart disease death (alive or not heart 0; heart death 1)

Now we'd like to start exploring the data in Splus, but first we need to learn a bit about how Splus stores data.

Data frames Splus has a nice way of packaging data into objects called 'data.frames'. Data frames are analagous to SAS data sets, in that they are a *collection of named columns of data* (like SAS variables), which all have the same length, and their rows are indexed by an identifier called '`row.names`' (like an observation number or ID variable in SAS). Different columns of data can have different data types (called *modes* in Splus), including numeric, factor, or ordered factor.

The `sas.get()` function The `sas.get()` function converts a SAS dataset into a data.frame. (It works by constructing and submitting a special SAS program which reads your SAS dataset, and then converts it into an annotated ASCII file, which is then converted into a data.frame via Splus commands.) Note that **you need to also have SAS running on your computer** in order to use the `sas.get()` function, and I think `sas.get()` only works under UNIX. Windows users have other alternatives for converting SAS data to Splus data.frames.
A few more things about `sas.get()`:

- `sas.get()` knows how to convert SAS missing values, i.e., '.', to the appropriate Splus missing value token, '`NA`', which stands for 'Not available'.
- `sas.get()` converts the underscores in variable names to dots.

Let's Create an Splus `data.frame` from an SAS dataset and check the working directory to see if our new object is there:

```
> hanes1_sas.get("/usr1/users/biostat/eeb271/datasets","hanes1")
> objects(1)
```

Now let's examine the `hanes1` object:

```
> class(hanes1)
> dim(hanes1)
> names(hanes1)
```

How many observations does the data.frame have? How many variables?

Now we can examine just one named column in the hanes1 `data.frame`. The numeric variables were stored by default as double precision, which is unnecessary for integer variables such as age. Changing the `storage.mode` to "integer" will reduce the amount of disk-space the object takes up. The storage mode may be altered by assigning a new string to the storage.mode of that variable:

```
> mode(hanes1$age)
> storage.mode(hanes1$age)
> storage.mode(hanes1$age)<-"integer"
```

Now, let's attach the data.frame to position 3, and then list the objects that it contains:

```
> attach(hanes1, pos=3)
> search()
> objects(3)
```

Note that we now have an object named 'age' on our search path, so instead of having to use the longhand 'hanes1$age', we can simply access that object via the shorthand 'age'. Please note that position 3 (where hanes1 has been attached) is not your working directory: if you modify any variables using the shorthand notation, the new versions will be stored in position 1. I recommend that you only make modifications to the variables using the longhand notation.
Here's an illustration of something very confusing which can happen if you don't follow my recommendation:

```
> find(age,numeric=T) # this tells us where age is on the search list
> age<-age/10         # let's modify age
> find(age,numeric=T) # now see where age is
> detach(3)           # this deletes the 3rd element on the search list
> find(age,numeric=T) # now see where age is
> hanes1$age[1:10]
> age[1:10]
```

Note: Comments can be made in Splus code using the # symbol; everything to the right of the # is a comment.
To reiterate: when using attach(hanes1), Splus *has made a copy* of the hanes1 dataset to a new path on the search list. If you want to modify any of the variables in the hanes1 object permanently, you must reference them using the longhand format: `hanes1$age`.

1.2.4 Data description functions

Calculate summary statistics for age and describe the distribution of age in this cohort:

```
> mean(age)
> median(age)
> quantile(age)
> quantile(age, c(0.1,0.9))
```

The summary() function provides a more efficient method for summary statistics.

```
> summary(age)
> page(summary(hanes1))
```

Create a categorical variable for age quintiles. This call to the `cut()` function chooses five equally spaced intervals of age:

```
> age5_cut(age, 5)
> table(age5)
```

This version of the cut() function chooses cut points that are quintiles of age.

```
> age5_cut(age, quantile(age, c(0,.2,.4,.6,.8,1)), include.lowest=T)
> table(age5)
> class(age5)
> summary(age5)
```

The categories have equal numbers of subjects, but are not equally spaced on the age axis. The `include.lowest` argument prevents the lowest values from being discarded.

1.2.5 Handling missing values

Examine the correlation of age with bmi. The following command will not run because bmi contains 3 missing values, as you should have see when you used the summary function before.

```
> cor(age,bmi)
> summary(bmi)
```

We can generate *logical vector* to use as an index to indicate which observations of data have a complete set nonmissing values for a set of variables. This logical vector may be incremented as necessary to account for additional variables.

```
> OK_!is.na(age) & !is.na(bmi) & !is.na(ser.chol) & !is.na(bp1sys)
> table(OK)
> OK_OK & !is.na(race) & !is.na(booze) & !is.na(sex) & !is.na(smokever)
> table(OK)
```

Remember the correlation that we could not run in the last exercise because of missing values? Now we can subset the bmi and age vectors to only those values with $\hat{O}$K==T. Note: Splus knows how to convert logical vectors to binary vectors, mapping `F -> 0` and `T -> 1`, so the following two commands are equivalent:

```
> cor(age[OK],bmi[OK])
> cor(age[OK==1],bmi[OK==1])
```

1.2.6 Basic plotting functions

First we need to set up a graphics device. Until we get Splus for Windows installed here, we are going to use a device which I set up that uses ghostview on Windows to display the graphs. Please do the following:

1. You need to make a folder in the C: drive on your Windows machine for downloading the graphics files.
2. Set the 'dowload directory' in Simpterm to be that folder.
3. Change the transfer option in Simpterm so that it overwrites files with name clashes, (This will enable your graphics to be updated automatically in your ghostview screen.)
4. Open ghostview in Windows; it is available from the 'Start' menu, under 'programs', 'gstools', 'gsview'.
5. I've set up a function called 'ps()' which sends your graphs (in postscript format) via kermit to the PC. You need to attach the library which has that function:

```
> attach("/usr1/users/biostat/eeb271/.Data")
```

Now, let's make a histogram for age:

```
> hist(age, nclass=5, plot=F)
> ps(hist(age, nclass=5))
```

The default name of the file created by 'ps()' is 'temp.ps', and you should open that file in the ghostview window. Your graph should appear in the ghostview window. You can use the ghostview menu to save your graph to a different filename, or print to the printer in the microlab.
Please note that if you are not in the microlab and are using one of the other graphics devices, you need to omit the ps() *call, and just submit:*

```
> hist(age, nclass=5)
```

1.2.7 Before you Leave

For those of you who are planning to do your homework on another computer, you will need to get the data onto that computer. Here's a way to save a portable version of the data which can be read into Splus on any platform.
First, here's how to dump the Splus dataset to an ASCII file. This ASCII dataset can now moved using ftp or floppy diskette to any platform including another Unix system or a PC.

```
data.dump("hanes1","hanes1.dmp")
```

The file you need transfer or put onto diskette is located in your current directory, and called "mydata.dmp".
Now, to restore the Splus dataset using Splus on the other computer,

```
> data.restore("hanes1.dmp")
> objects(1)
```

And your data should have been converted back into an Splus data.frame object.

1.3 Homework for Week 1

Please note that you may need to review/repeat some of the exercises in lab 1 in order to get your terminal set up properly to display graphics.

1. Compute quantiles and boxplots for serum cholesterol:

   ```
   > median(ser.chol)
   > quantile(ser.chol)
   > quantile(ser.chol, c(0.1,0.9))
   > boxplot(ser.chol)
   > boxplot(log(ser.chol))
   ```

   What is the distribution of serum cholesterol? Does the distribution of log(ser.chol) look closer to the normal distribution?

2. Does serum cholesterol vary with age? Describe the apparent relationship of serum cholesterol with age as shown by a scatter plot.

   ```
   > plot(ser.chol, age)
   ```

3. Plot the distribution of systolic blood pressure for each category of booze. Booze is a numeric vector with 0 for the lowest level of alcohol consumption and 3 for the highest.

   ```
   > boxplot(split(bp1sys, booze), notch=T)
   > boxplot(split(bp1sys, booze), varwidth=T)
   ```

   Which plot is most informative regarding the relationship of these variables? Using this same strategy, now make a boxplot for serum cholesterol by age quintiles.

4. In the previous questions, you've looked at the variable serum cholesterol in several ways. Please synthesize your findings in a short 2-5 sentence paragraph that would be suitable for publication.

2 Week 2

2.1 Preparation for Week 2: Functions for Generalized Linear modelling in Splus

2.1.1 Arguments to the `glm()` function

The Splus function `glm()` fits a generalized linear model, and takes the following arguments:

formula This argument should be an object of class *formula* specifying the model you'd like to fit. To construct a formula object, you simply write something like `y~x1+x2`, that is, put your response variable on the left hand side of `~`, and a modelling expression on the right hand side of `~`. For GLM modeling, this is understood as "y is modelled as a linear combination of the variables `x1` and `x2`". An intercept is included by default. To remove the intercept, you can simply subtract '`1`' from your expression, i.e., `y~x1+x2-1` specifies a no-intercept model.

A really nice feature of Splus is that you can create transformations of your covariates right in the formula expression (there is no need to construct them ahead of time in your data frame). For example, say you wanted to use the square of `x1`, ratio of `x2/x3`, and the sum of `x4` and `x5` as covariates in your model. This could be accomplished with the following formula: `y~I(x1^2)+I(x2/x3)+I(x4+x5)`. (Here the wrapper function `I()` is needed to 'quote' operators so they are interpretted as math operators rather than formula operators.)

Please note that the formula argument is the only required argument to `glm()`. For the rest of the arguments, if nothing is specified, default arguments will be in effect. Also, the optional arguments below are usually specified by giving the argument name, then an '=' sign, then the argument value.

family This argument should be the name of a function used to define the error distrubution family. The following table shows choices for the error distribution family which are provided with Splus, as well as the default (and other possible) link functions:

Family	Link Function $g(\mu) = \eta = \beta^T X$	Variance Function $var(y) = \phi V(\mu)$
Gaussian	identity	1
Binomial	logit, (probit, cloglog)	$\mu(1-\mu)/n$
Poisson	log, (identity, sqrt)	μ
Gamma	inverse, (identity, log)	μ^2
Quasi	$g(\mu)$	$V(\mu)$

For example, to specify the family for a logistic regression (with default link), use `family=binomial`. Alternatively, for probit regression, you can use `family=binomial(link=probit)`.

The default action of `glm()` is to do Normal (Gaussian) regression.

data This argument should contain either the name of a data frame, or an expression which results in a data frame. By default, Splus will look for variables in your `search()` list if no data argument is given.

subset this argument should contain an expression which results in a logical vector which can be used to subset your data frame. The variables used in the expression should be in your data frame. The default subset is the complete data frame.

na.action this argument should specify a function which is called on the data frame when you are missing any of the variables which you need for this particular call to glm. An often used specification for this argument is `na.omit`, which will automatically omit any observations that have missing values from your data frame. (In doing so, only considers the variables referred to in the call.) This feature is flexible enough to allow a user to write their own function to impute missing valeus. The default action, `na.fail`, is to "fail" to do the fit and return an error message if any missing values are present.

Here is an example of how you might create a fit object for a logistic regression of death from ischemic heart disease modelled on age and body mass index for females in the NHANES1 dataset:

```
> fit1 <- glm(d.heart~age+bmi,
              family="binomial",
              data=hanes1,
              subset=sex==0,
              na.action=na.omit)
```

2.1.2 Components of the glm fit object

The object resulting from a call to `glm()` is called a 'fit.object', and has many components [5].

First, here are the components which are used for describing features of the model:

call a (recursive) object which contains the 6 arguments supplied to the `glm()` call: a formula expression, the data.frame, a subset expression, a character string specifying the error distribution family, a list specifying the contrasts, and an action for missing values.

formula an object that contains the formula expression that created the glm object.

rank the number of regression coefficients (p), provided that the model is full rank.

df.residuals the number of degrees of freedom, $(n - p)$.

terms an object of length 3 that contains information regarding the terms in the formula.

family a character vector of length 3 that provides the name, link, and variance arguments for the error distribution family.

There are also some components which are used for storing features of the fitting results:

coefficient a numeric vector of length p with regression estimates (i.e., the β's).

null.deviance the deviance statistic under the null hypothesis, H_0.

deviance the deviance statistic under the alternative hypothesis, H_A.

iter the number of iterations that were used to provide the model fit.

R a numeric upper triangular matrix of dimension $p \times p$ that contains the correlation coefficients for the regression estimates.

2.1.3 Functions which extract information from the fit object

Fortunately, Splus provides several *methods* for extracting and displaying the information stored in the object fit, so it's not something you as a data analyst will usually need to keep track of. Here are some functions which can be used to extract some of the model components (all of these functions take a fit object, such as `fit1` in the above example, as an argument):

formula() extracts the formula from a fit object

[5] An easy way to display the names of the components of an objects is to use the `names()` function to display them, i.e., in the example above, `names(fit1)` will return a vector of character strings indicating component names.

family() extracts a complete specification of the error distribution family and link function, which is printed [6] in a concise informative format.

coef() extracts the estimated regression coefficients, a numeric vector with length p.

deviance() extracts the deviance (that is, the deviance under H_A; this is NOT the `null.deviance`).

Splus also provides some functions for displaying the results of the fit:

summary(`fit.object,c=F`) The `summary` function [7] summarizes all of the regression results contained in a glm object. The returning values are described below for the summary glm object. An object with class `"summary.glm"` is returned, and contains, most noteably, information about the covariance of the regression estimates and the dispersion. The `c=F` argument suppresses the output of the correlation matrix for the coefficients.

summary.aov(`fit.object`) summarizes the regression results for a series of nested models with one covariate removed from the model in the glm object.

predict(`fit.object, "type"`) This function calculates predicted values. The arguments include a glm object and a type of prediction. Possible types include:

`"response"` for predictions on the same scale as the original dependent variable, and

`"link"` for predictions on the scale of the link transformation.

If the arguments include a new data matrix of covariate values with dimensions $n \times p$, the function returns a numeric vector of length n with the predicted values for the dependent variable.

fitted(`fit.object`) This function extracts the predicted values on the same scale as the original dependent variable.

residuals(`fit.object, "type"`) This function extracts the residuals from the fitted model. The `type` argument specifies any one of:

`"deviance"` which is the default, and equivalent to `summary(fit.object)$deviance.resid`,

[6] What is actually returned is an object that has class 'family', and there is a print method for family which displays the contents of this object in a nice format, listing the family, link and variance function. However, the actual object returned has 9 or more highly specific components needed by the glm fitting procedure to construct a model belonging to a particular family. This returned object can actually be used as a template for making a modified version of the family.

[7] Technically, the `summary()` function is a generic function which calls `summary.glm()`, the specific *method*, for objects which have class `"glm"`.

`"working"` which is equivalent to `fit.object$residuals`,

`"pearson"` a weighted version of the working residuals, or

`"response"` which are on the same scale as the dependent variable.

lm.influence(`fit.object`) This creates an object with 3 components concerning the influence diagnostics of the fit:

lm.influence(`fit.object`)`$coefficients` a numeric matrix with dimensions $n \times (p+1)$ that contains for each observation the regression coefficients with that observation removed.

lm.influence(`fit.object`)`$sigma` a numeric vector with length n that contains for each observationthe estimated residual error in the model with that observation removed.

lm.influence(`fit.object`)`$hat` a numeric vector with length n that contains the diagonal elements of the hat matrix that maps the response, y, onto $\hat{y}$.

2.2 Laboratory for Week 2: GLM Example

2.2.1 More setup stuff

Getting the new dataset

1. Last week we used the wrong version of the NHANES dataset. Remove the old version, and get the new version. You will also need to recreate that `OK` logical vector for indicating non-missing values that we created last week.

```
> rm(hanes1)
> hanes1a <- sas.get("/usr1/users/biostat/eeb271/datasets","hanes1a")
> OK_!is.na(age) & !is.na(bmi) & !is.na(ser.chol) & !is.na(bp1sys)
> OK_OK & !is.na(race) & !is.na(booze) & !is.na(sex) & !is.na(smokever)
```

Setting up your graphics environment the microlab

2. Run the following script at the UNIX prompt, and invoke the changes:

```
hsph% /usr1/users/biostat/eeb271/setup
hsph% source ~/.cshrc
```

You only need the above commands once. This will permanently set-up 3 things for you, so that next time you log in they will be available to you: the line `setenv S_CLEDITOR emacs` will be added to your `.cshrc`, special kermit transfer scripts will be add to your `.kermrc`, and a function `.First` will be created in your Splus working directory.

There are some things you still need to do *every time* you use Splus in the microlab if you want to display graphics:

- in the telnet menu, go to the transfer menu, and choose transfer setup. *un*check the box that says something like "rename duplicate files".
- start the gstools application. It can be accessed somewhere inside the start-programs menu.

Functions for editing Splus objects

3. We can also create our own version of the editor function, `ed()`, to facilitate editing Splus objects using your favorite text editor. The UNIX editor 'pico' is the same easy-to-use editor that you've probably been using to compose email in pine. Here's how to create a `pico()` function that you can use to edit Splus objects:

```
> pico <- function(object)ed(object, editor = "pico")
```

4. When you have a function called `.First` in your Splus working directory, it will be automatically executed every time you start Splus. Now, use the `pico()` function to edit your `.First` function to add commands you need to run every time you use Splus. (Right after you modify the `.First` function, you should run it to have the new items take effect during the current Splus session.)

```
> .First <- pico(.First)
> .First()
```

Now, you can add some of the following lines to your `.First` function definition:

```
options(contrasts=c("contr.treatment","contr.poly"))
options(editor="pico")
attach(hanes1a)
```

2.2.2 Linear models for systolic blood pressure using `glm()`

5. Create a simple regression model for blood pressure among men. To extract the male subset of `hanes1a`, we need to create a logical vector `sex==0` which has value `T` for males, and `F` for females. Describe the association between blood pressure and each independent variable. What assumptions have we made about the relationship of blood pressure and these predictors?

```
> bp.fit <- glm(formula=bp1sys~age+smokever+bmi, family=gaussian,
+ na.action=na.omit, data=hanes1a, subset=(sex==0))
> summary(bp.fit,c=F)
```

- Aside: Are you wondering what that `c=F` argument meant in the call to `summary`?
  Go find out for yourself:
  - `> help(summary)` # will explain that `summary` is a generic function, the specific function (or 'method') used will be chosen based on the class of the argument.
  - `> class(bp.fit)` # tells you that the class of the fit is `"glm"` which inherits from `"lm"`.
  - `> ?summary.glm` # will explain what the argument `c=F` meant.

6. Alter bmi to a quadratic function and update the regression. Does the quadratic term improve the fit of the model? How does the linear bmi term contribute to the model?

   Note: You may use either the update function as below or recall (using (in windows or `Ctrl-P` in UNIX) and edit the previous model statement. If you recall the previous model, don't forget to change the name of the output object as well as the formula.

```
> bp.fit2 <- update(bp.fit,.~.+I(bmi^2))
> summary(bp.fit2,c=F)
> anova(bp.fit, bp.fit2, test="F")
```

7. Delete the linear bmi term. How does this alter the model? What assumption has been made about the quadratic function in this model?

```
> bp.fit2 <- update(bp.fit, . ~ . - bmi)
> summary(bp.fit2,c=F)
```

8. Update to add alcohol as a risk factor. The as.factor() function forces Splus to treat booze as categorical variable. Decribe the association of alcohol and blood pressure.

```
> bp.fit <- update(bp.fit, . ~ . + as.factor(booze))
> summary(bp.fit,c=F)
```

Effect modification by gender

9. Repeat for women: `subset=(sex==1)`. Is gender an effect modifier for any covariate?

```
> bp.fit2 <- update(bp.fit, subset=(sex==1))
> coef(bp.fit)
> coef(bp.fit2)
```

2.2.3 Preliminary plots and statistics for mortality

10. What is the cumulative incidence of total mortality, ischemic heart disease mortality, and cancer mortality in this cohort?

```
> mean(d.total)
> mean(d.heart)
> mean(d.cancer)
```

11. Calculate the cumulative incidence of heart disease by gender and by age quintiles. Does this meet your prior knowledge regarding these variables?

```
> tapply(d.heart, sex, mean)
> tapply(d.heart, age5, mean)
```

12. Plot the distributions of age and serum cholesterol for the survivors and the dead for total mortality. How do these distributions differ in each case? Repeat for ischemic heart disease (`d.heart`) and cancer (`d.cancer`). Are the distributions of age and serum cholesterol for these outcomes similar to those observed for all cause mortality?

```
> boxplot(split(age, d.total), notch=T)
> boxplot(split(ser.chol, d.total), notch=T)
```

2.2.4 Joint distribution of serum cholesterol, age, and cause of death

13. Create numeric vector of the cumulative incidence of death by age quintile for total mortality, ischemic heart disease mortality, and cancer mortality.

```
> p.total <- tapply(d.total, cut(age,5), mean)
> p.d.cancer <- tapply(d.cancer, age5, mean)
> p.d.heart <- tapply(d.heart, age5, mean)
```

14. Plot probability of death vs. probability of death due to specific cause

```
> plot(p.d.heart, p.total)
> plot(p.d.cancer, p.total)
> plot(p.d.cancer, p.d.heart)
```

2.2.5 Logistic regression model for total mortality

15. Create a logistic regression model for mortality with main effects for age and sex. Explain the meaning of each coefficient. Note that for the binomial error distribution the default link is logit for a logistic model. The alternative links are probit and complementary log-log.

```
> fit <- glm(d.total~sex+age, data=hanes1a, family=binomial,
+ na.action=na.omit)
> summary(fit, c=F)
```

16. Instead of the na.action, we could have used the subset=(OK==1) argument to restrict the dataset to observations with complete information on a list of potential covariates. Use the `anova()` function to compare these two models. What is wrong with this comparison?

```
> fit2 <- update(fit, subset=OK)
> summary(fit2, c=F)
> anova(fit2, fit, test="Chi")
```

2.3 Homework for Week 2

2.3.1 Checking for effect modification

1. Continuing from item 16 in this week's laboratory, introduce a term for the interaction of age and sex using either method. Explain the meaning of each coefficient. How does the interaction term effect the interpretation of the coefficients for sex and age? Does the interaction term significantly improve the overall fit of the model?

```
> fit2 <- update(fit, . ~ . + I(sex*age))
> fit2 <- update(fit, . ~ sex * age)
> summary(fit2, c=F)
> anova(fit2, fit, test="Chi")
```

2.3.2 Logistic model for ischemic heart disease

2. Construct a logistic model for ischemic heart disease (`d.heart`) by substituting ischemic heart disease deaths (`d.heart`) as the dependent variable, and then add serum cholesterol to the model. Does cholesterol improve the overall fit of the model?

```
> fit <- update(fit, d.heart ~ .)
> summary(fit, c=F)
> fit2 <- update(fit, . ~ . + ser.chol)
> summary(fit2, c=F)
> anova(fit2, fit, test="Chi")
```

Also, complete the following steps, and summarize your findings in a paragraph that would be suitable for publication:

a) Add systolic blood pressure (bp1sys) to the logistic model for ischemic heart disease.

b) Add body mass index (bmi) to the preceding model.

c) Drop systolic blood pressure (bp1sys) from the preceding model.

d) Substitute total deaths (d.total) as the dependent variable in our revised model.

e) Substitute cancer deaths (d.cancer) as the dependent variable in our revised model.

f) Add a quadratic term for serum cholesterol to the previous model for total deaths.

3 Week 3

3.1 Preparation for Week 3: Advanced Parametric Models

3.1.1 Formula expressions

offset(numeric.vector)
used in the formula expression for a Poisson regression. The numeric vector in the argument is entered as a linear predictor, but no coefficient is fitted.

cut(numeric.vector, 5)
creates a factor vector that specifies the category (1-5) for each element of the numeric vector. The cutpoints for the categories are evenly spaced across the range of the numeric vector and the cutpoint values are stored as a names vector.

cut(numeric.vector,quantile(numeric.vector, c(0,.2,.4,.6,.8,1)),include.lowest=T)
creates a factor vector that specifies the quintile (1-5) for each element of the numeric vector. The cutpoints for the quintiles are stored as a names vector.

I((numeric.vector > cutpoint)*(numeric.vector - cutpoint))
used in piecewise regression models, this function creates a new numeric vector that is zero at or below the cutpoint and increases linearly after the cutpoint. The original numeric vector should be retained in the model unless the effect is truly zero below the cutpoint.

3.1.2 Partial residual plots

residuals(update(model.object, .~. - predictor))
creates a vector of partial residuals of the dependent variable with one variable removed from the model but the other independent variables are still in the model.

mean(predictor)+residuals(update (model.object, predictor ~. - predictor))
creates a vector of partial residuals of one predictor variable with the other independent variables still in the model. For publication purposes, it is useful to add the mean to the residuals to place them on the natural scale, so that the plot does not have negative values of the predictor variable.

tapply(residuals, cut(residuals, quantile(residuals,c(0,.2,.4,.6,.8,1)), include.lowest=T),mean)
creates a numeric vector with the adjusted means for the quintiles of the numeric vector of residuals. For a partial regression plot of a categorical predictor, these residuals would be the partial residuals for that predictor variable.

3.1.3 Orthogonal polynomial functions

poly(numeric.vector,degree)
specifies a polynomial expansion of the numeric vector. This saves typing the polynomial terms (e.g. `poly(age, 3)` instead of `age + I(age^2) + I(age^3)`). However the `poly()` function finds a set of three cubic polynomials with fixed coefficients that are orthogonal in your data and uses them instead of the `age`, `age^2`, and `age^3` terms. The poly function has two advantages: all of the components of age may be plotted as a single dose-response curve using `plot.gam()` and the orthogonal transform reduces numerical error in the regression estimation.

poly.transform(poly(numeric.vector, degree),coef(model.object)[-c(list)])
reexpresses the coefficients of an orthogonal polynomial in terms of a simple power series. The polynomial must be stated as it appears in the model object. The list argument must be used to suppress the coefficients for any other independent variables in the model.

3.1.4 Spline functions

In the following spline functions, the number of knots are specified by a df argument. The knots are placed at suitably chosen quantiles of the independent variable. Alternatively, the user to may directly specify the location of the knots using a knots=list argument.

Truncated power basis: This is the most intuitive way to specify a spline model using a piecewise polynomial fit, specified as:

```
age + I(age^2) + I(age^3) + I((age > age0) * ((age-age0)^3))
```

for one knot at age0 (df = 4), and similarly for additional knots.

Alternatively, Splus will fill in the spline functions for you using orthogonal transformations to reduce numeric error in computations. As with the `poly()` function, using an Splus spline function will permit you to easily plot the dose-response function.

Two commonly used forms are B-splines, using `bs()`, and natural (or restricted) splines, using `ns()`. The difference between `bs()` and `ns()` is that `ns()` is restricted to be linear at the extremes of the data, where otherwise the lack of data could produce wild results. We recommend the use of natural splines for this reason.

bs(numeric.vector, df=n, degree=3)
Specifies a piecewise-cubic spline used in the formula expression. The numeric vector specified as an argument is converted to a piecewise-cubic spline and the returning numeric vectors are entered as independent variables. The default degree of 3 specifies a cubic spline model.

ns(numeric.vector,df=n)
Specifies a natural (or restricted) cubic spline used in the formula expression. The numeric vector specified as an argument is converted to a natural cubic spline and the returning numeric vectors are entered as independent variables.

3.1.5 Plotting Functions

par(mfrow=c(rows, columns))
Sets up the graphic output for multiple plots on one page.

plot.lm(model.object, ask=F)
Creates six diagnostic plots for linear models.

plot.glm(glm.model.object, ask=F)
Creates four diagnostic plots for generalized linear models. The six diagnostic plots for linear models are still available for glm models using the `plot.lm()` function.

lowess(numeric.vector, numeric.vector)
A simple nonparametric smoothing function of the association between the two numeric vectors using robust locally linear fits. The default window includes two-thirds of the data and may be adjusted using an optional `f=` argument.

plot.gam(glm.model.object, ask=T)
Plots the dose response relationships with the independent variables. This is particularly useful for categorical, polynomial, and spline variables. For categorical variables (age5 or booze), the plot provides the response for each category. For polynomial or spline functions, each plot provides the combined response function for the multiple regression coefficients associated with an independent variable.

3.2 Laboratory for Week 3: Parametric smoothing example

3.2.1 Linear model for blood pressure among men

1. If you are not sure if your OK variable is current, you can try recreating it using a new function which I made and put in the "/usr1/users/biostat/eeb271/.Data" directory which you should have attached.

```
> OKvars # a character vector of names of variables you want to be nonmissing
> OK_makeOK(OKvars) # make your OK vector
> table(OK) # see how many observations will be subsetted
```

   Let us return to our linear model for blood pressure.

```
> bp.fit_glm(bp1sys~smokever+age+bmi, family=gaussian,
+    subset=OK & female, data=hanes1a)
> summary.lm(bp.fit,c=F)
```

3.2.2 Diagnostic plots for linearity

2. We can examine the assumption that the relationships between blood pressure and the continuous variables (age and bmi) are linear using *partial regression plots.*
   Reminder: a partial regression plot for any covariate X has a

   - y-axis of residuals of the model with X removed, and an
   - x-axis with the residuals of from modelling X on all the remaining covariates.

   Here we have added a constant back to our x-axis corresponding to the mean of the X variable for interpretability.

```
> r.drop.age_residuals(update(bp.fit, .~. -age))
> regadj.age_mean(age[OK &female])+residuals(update(bp.fit,age~.-age))
> plot(regadj.age,r.drop.age)
> lines(lowess(regadj.age,r.drop.age))
```

   Are the residuals evenly distributed across the range for age?

3. Repeat for body mass index. Are the residuals evenly distributed across range for bmi?

```
> r.drop.bmi_residuals(update(bp.fit, .~. -bmi))
> regadj.bmi_mean(bmi[OK &female])+residuals(update(bp.fit,bmi~.-bmi))
> plot(regadj.bmi,r.drop.bmi)
> lines(lowess(regadj.bmi,r.drop.bmi))
```

3.2.3 Robust regression

4. How are the more extreme residuals affecting the fit of the regression model? Switch to a robust(gaussian) error distribution to reduce the influence of the most extreme residuals. (See page 259 in V&R for a more thorough description of the `robust` family generator for `glm`.)

```
> bp.fit2_update(bp.fit, family=robust(gaussian))
> summary.lm(bp.fit2, c=F)
> anova(bp.fit2, bp.fit)
> cbind(normal=coef(bp.fit),robust=coef(bp.fit2))
```

   Is there a problem with this ANOVA?
   How do the new coefficients differ from those for the previous model?

   If you have time, you might want to try redoing the plots in items 2 and 3 for this model to see how the residuals are now "tightened" up.

3.2.4 Consider a nonlinear transformation using standard methods

I've created a quintiles function so that we won't have to do so much typing. It's located in the eeb271 .Data directory which should already be attached.

5. Create categorical vectors for age and bmi quintiles. Replace the linear terms for age and bmi with the quintiles. How do the new results differ from those for the linear model?

```
> age5_quintiles(age,OK & female)
> bmi5_quintiles(bmi,OK & female)
> bp.fit2_update(bp.fit, bp1sys ~ smokever + age5 + bmi5)
> summary.lm(bp.fit2,c=F)
> anova(bp.fit2, bp.fit, test="F")
> bp.fit3_update(bp.fit, bp1sys ~ smokever + numeric(age5) + numeric(bmi5)
> anova(bp.fit2, bp.fit3, test="F")
```

   Which of the 2 ANOVAs gives a valid inference?

6. Plot coefficients by quintile means. What are the shapes of these associations?

```
> cbind(coef(bp.fit2)) # a neat way to transpose the coef vector
>                      # so we can see what position the betas are in
> mean.age.q5_tapply(age, age5, mean)
```

```
> plot(mean.age.q5, c(0, coef(bp.fit2)[3:6]))
> mean.bmi.q5_tapply(bmi, bmi, mean)
> plot(mean.bmi.q5, c(0, coef(bp.fit2)[7:10]))
```

7. Plot coefficients vs. means of quintiles of regression adjusted covariates. What are the shapes of these associations?

```
> r.age5_mean(age[OK & female]) + residuals(update(bp.fit2, age~. - age5))
> plot(tapply(r.age5, quintiles(r.age5), mean), c(0, coef(bp.fit2)[3:6]))
> r.bmi5_mean(bmi[OK & female]) + residuals(update(bp.fit2, bmi~. - bmi5))
> plot(tapply(r.bmi5, quintiles(r.bmi5), mean), c(0, coef(bp.fit2)[7:10]))
```

8. The plot for age looks somewhat linear while the plot for bmi looks nonlinear. Fit a piecewise regression for bmi with an arbitrary breakpoint at 28.

```
> bp.fit3_update(bp.fit, . ~ . + I((bmi > 28) * (bmi - 28)))
> anova(bp.fit3, bp.fit, test="F")
```

9. Replace the linear terms for age and bmi with second degree polynomials. Does this improve the model's fit? What are the coefficients for the linear and quadratic terms for age and bmi?

```
> bp.fit2_update(bp.fit, . ~ smokever + poly(age,2) + poly(bmi,2))
> anova(bp.fit2, bp.fit, test="F")
> summary(bp.fit2, c=F)
> poly.transform(poly(age,2), coef(bp.fit2)[-c(2,5,6)])
> poly.transform(poly(bmi,2), coef(bp.fit2)[-c(2,3,4)])
```

10. Use the plot.gam() function to plot the polynomial functions for age and bmi. Hint: Turn off the rug plot and turn on the se plot. What are the shapes of these polynomial functions? Now turn on the residual plot. Are the new residuals more evenly distributed across the ranges of age and bmi? How do the new residuals compare with the old residuals?

```
> par(mfrow=c(2,2))
> plot.gam(bp.fit2,rugplot=F,se=T)
> plot(bp.fit$residuals, bp.fit2$residuals)
```

3.3 Homework for Week 3

Please attempt to answer the required questions using Splus modelling techniques that you learned in Lab 3. It would be helpful if you could also provide your Splus source code and any graphics produced so that I can help troubleshoot in case something went wrong!

3.3.1 Logistic model for total mortality among women

Let us return to our logistic model for total mortality. Set up your `OK` variable so that you can subset to observations which have the following variables nonmissing:
`age, race, smokever, bmi booze, bp1sys.`

```
> fit <- glm(d.total ~ age+race+smokever+bmi+as.factor(booze)+bp1sys,
+    data=hanes1a, family=binomial, subset=OK & female)
```

3.3.2 Diagnostic plots for linearity

1. Examine the assumption of a linear relationship between mortality and each continuous variable (age, bmi, bp1sys) using partial regression plots. Do the residual plots show a linear relationship with age and blood pressure? Note: don't forget to use the lowess() command to show the trend in your scatterplot.

3.3.3 Robust regression

2. How do the more extreme residuals affect the fitting of the regression model? We can reduce the influence of the most extreme residuals using a robust(binomial) error distribution. How do the new results differ from those for the previous model?

3.3.4 Spline models for age

3. Develop polynomial and natural spline models (see function `ns()` in the Preparation for Lab 3) for the association of age with total mortality. What are the differences between these models? Which model provides the best fit to the data? You may want to experiment with other choices for degrees of freedom.

4. Plot the smoothed relationship between age and the predicted values from each modeling technique. Discuss the shape of the association between age and mortality.

5. **Optional:** Switch the model to cancer deaths. What is the association of age with cancer mortality? Is this association linear? Which parametric smoothing function

provides the best fit to the data? Are there any "bumps" in the smooth curve? Are these biologically plausible? Which parametric smoothing function shows this nonlinearity the best?

6. **More optional exercises, drawing from Louise's Feb. 16 lecture:**
Generate some data to demonstrate that X and X^2 are correlated.

```
> for(i in 1:10) # simulate this 10 times
+ {
+    x_runif(1000)
+    print(cor(x,x*x))
+ }
```

Now, show that X and $(X - E(X))^2$ are less correlated:

```
> for(i in 1:10)
+ {
+    x_runif(1000)
+    print(cor(x,(x-mean(x))^2))
+ }
```

Now, use `poly()` to construct orthogonal polynomials corresponding to the linear and quadratic terms, and show that they are uncorrelated:

```
> x_runif(1000)
> xpoly2_poly(x,2)
> cor(xpoly2[,1],xpoly2[,2])
```

4 Week 4

4.1 Laboratory for Week 4: UNIX

4.1.1 Some UNIX commands

(We covered some additional UNIX commands including cp, ls, kermit, etc. during lab. I will try to put notes on this here.)

4.1.2 Splus source code

```
# sample(x, size=<<see below>>, replace=F, prob=<<see below>>)
# Use the sample function to randomly sample "size" number
# of values from possible values listed in x.  Use replace=T
# to sample with replacement.  use the prob argument to specify
# probabilities corresponding to each value in x.
#
# list the default args of the sample function:
args(sample)

# simulate a dichotomous rare outcome:
y_sample(0:1,1000,T,p=c(.95,.05))

# simulate a dichotomous covariate which is about equal in the population:
x1_sample(0:1,1000,T,p=c(.5,.5))

#
table(y,x1)

# make a function which will compute the odds ratio from
# a 2x2 table
OR_function(mat)mat[1,1]*mat[2,2]/mat[2,1]/mat[1,2]

# apply the function to our 2x2 table
OR(table(y,x1))

# now compute the log(odds)
log(OR(table(y,x1)))

# the beta for x1 in a logistic regression should give us the same log odds:
glm(y~x1,family=binomial)
```

```
# now let's make a continuous covariate, like age:
x2_rnorm(1000,25,5)
mean(x2)
var(x2)

# what if we run a least squares regression with our binary outcome?
summary.lm(lm(y~x1+x2),c=F)

# what if we run a generalized linear gaussian regression
# with our binary outcome? should be *exactly* the same as above...
summary.lm(glm(y~x1+x2),c=F)

# Now, what if we run a generalized additive (gaussian) model
summary.lm(gam(y~x1+x2),c=F)


# why are all these the same? because they all use least squares regression
# (a non-iterative technique, with a unique solution)

# Logistic regression can only be run in glm or gam.  These methods have
# different convergence criteria, so will give slightly different
# fits to the same model (assuming only parametric terms, since
# glm will only do those).
summary.lm(glm(y~x1+x2,family=binomial),c=F)

summary.lm(gam(y~x1+x2,family=binomial),c=F)

# These two should have the same coefficients for beta of x2:
glm(residuals(glm(y~x1))~residuals(glm(x2~x1))-1)
glm(y~x1+x2)

# does this property hold for logistic regression?
glm(residuals(glm(y~x1,family=binomial))~residuals(glm(x2~x1))-1)
glm(y~x1+x2,family=binomial)
```

4.1.3 Splus output

4.1.4 Running Splus in BATCH mode

Sometimes it's easier to put all of the Splus commands you'd like to submit for a particular exercise into a text file, and then submit the commands all at once to Splus. When you submit the file via BATCH mode, Splus will produce an output file which has each command followed by its result, much like you'd see on the screen if you were working interactively.

To submit a command file, say ''`mylab.q`'' to BATCH mode, and put the results in the file ''`mylab.out`'', you can type the following at the **hsph%** command prompt

```
Splus BATCH mylab.q mylab.out
```

Then, you can view the output file via "`less mylab.out`" on the hsph computer. You may also transfer this output file to your PC via kermit, and then incorporate it into a wordprocessor for further annotation and editing.

In my opinion, this method works better than sinking and echoing (which we learned about in Lab 1), as it produces more readable, self-contained output. Also, you can set the postscript graphics device at the beginning of the file, and then all the graphs that you produce in the BATCH session will be written to a postscript graphics file which you can transfer via kermit to your PC for viewing with GSview.

The homework 1 solution and lab3 output handouts are nice examples of the results of running a program in BATCH mode.

5 Week 5

5.1 Overview for Week 5: Nonparametric smoothing example

5.1.1 Smoothing Functions

These are simple bivariate, nonparametric smoothing functions. These bivariate smoothing functions do not adjust for any other covariates. The returning value for both functions is a list object containing an x-vector of sorted x values at which the kernel estimate was computed and a y-vector of smoothed estimates for the regression at the corresponding x. Note that the returned x-vector is a sorted version of the input x-vector, with duplicate points removed.

ksmooth`(x.vector, y.vector, kernel="box", bandwidth=0.5, n.points=length(x))`

Performs scatterplot smoothing using kernel estimates. The smoothing kernel is determined by one of the following character strings:

`"box"`	a rectangular box (the default)
`"triangle"`	a triangle (a box convolved with itself)
`"parzen"`	the Parzen function (a box convolved with a triangle)
`"normal"`	the Gaussian density function

Kernels are scaled so that their upper and lower quartiles (viewed as a probability density) are ± 0.25 when bandwidth is 1. Larger values of bandwidth make smoother estimates, while smaller values make less smooth estimates. The argument `n.points` specifies the number of points to smooth in the interval `range.x`, and the default is to smooth at each `x` value.

lowess`(x.vector, y.vectr, f=2/3)`

Gives a robust, local smooth of scatterplot data. A window, dependent on `f`, is placed about each `x` value; points that are inside the window are weighted so that nearby points get the most weight. The weights are determined by a tri-cubic function of the distance from each evaluated x-point for each of the other x-points within the window. The larger the `f` value, the smoother the fit.

5.1.2 GAM functions

Model Creation and Development

gam`(formula,family,data.frame,subset=(logical expression),na.action=na.omit)`

Creates a gam object. The arguments are the same as for glm(), but the formula expression may include smoothing functions of the independent variables.

lo(numeric.vector, span = f)
A lowess smooth used in the formula expression. The numeric vector is entered as a smoothed independent variable. The span is entered as a decimal fraction (f) for the portion of the range of the independent variable to be included in the smoothing window.

update(model.object,new.arguments)
Updates a gam object in the same manner that this function updates glm objects.

Extracting Model Components The functions for extracting glm model components may be used for gam model components.

5.1.3 GAM objects

Model description GAM objects contain all of the descriptive components of a glm object including formula, family, and terms.

Model results GAM objects contain all of the result and residual components of a glm object including coefficients, deviance, correlation coefficients, and residuals.

Instead of the `linear.predictors` we had in the glm object, we have `additive.predictors` in a gam object. For linear independent variables, the `additive.predictors` are the same as the `linear.predictors` in the glm object. For smoothed independent variables, the additive predictors are the linear portion of the smoothed variables' contribution to the estimated fit.

Gam objects also contain several components that are not found in glm objects. The size of these components depends on the number of smoothed independent variables in the gam mode, and these components are omitted when there are no smooth terms in the gam model.

gam.object$nl.df —a numeric vector giving the approximate nonlinear degrees of freedom for each smoothed independent variable. The total degrees of freedom for a smoothed independent variable will be 1 linear degree of freedom for its linear contribution plus the nonlinear degrees of freedom for its nonlinear contribution to the fit.

gam.object$nl.chisq —a numeric vector giving a type of score test for the removal of the nonlinear portion of each smoothed independent variable.

gam.object$smooth —a numeric matrix giving the smoothing function for each smoothed independent variable. An individual smooth may be extracted by specifying a column of this matrix: `gam.object$smooth[n,]`.

gam.object$var —a numeric matrix giving the approximate pointwise variances for the smoothing function for each smoothed independent variable. The variances for an individual smooth may be extracted by specifying a column of this matrix: `gam.objct$var[n,]`.

5.1.4 Summarizing model results

Summary Functions

summary.glm(gam.object,c=F)
Summarizes the linear portion of the regression results contained in a gam object. The returning values are described above for the summary glm object. The `c=F` argument suppresses the output of the correlation matrix for the coefficients.

summary(gam.object)
Summarizes the nonlinear portion of the regression results contained in a gam object. The returning values are described below for the gam summary object.

plot.gam(gam.object,se=F,rug=T)
Plots the relationship between the dependent variable (as the deviation from the mean on the y-axis) and each independent variable (natural scale on the x-axis). For linear independent variables, the plots will be straight lines. For smoothed independent variables, the plots will provide the relationship of the two variables over the range of the independent variable. The `se` argument controls the output of approximate point-wise 95% confidence bounds around the smooth. The `rug` argument controls the output of a rug plot above the x-axis describing the distribution of observations over the range of the independent variable. The rug plot may considerably increase the plotting time on some graphics devices.

Summary Object A recursive object that may be created by applying the summary function to a glm object. The returning value of the summary function may be stored as an object or listed as text on the default output device.

summary.object$call, summary.object$terms, and summary.object$iter
These are the same as for the similarly named elements of the glm object. The complete function call and the number of iterations are supplied in the listed output.

summary.object$null.deviance and summary.object$deviance
These are the same as for the similarly named elements of the glm object and both are supplied in the listed output.

summary.object$coefficients
A numeric matrix with dimensions $p \times 3$ that contains the estimate, the standard error of the estimate, and the t-statisti for each regression coefficient. This matrix is supplied in the listed output.

summary.object$df
A numeric vector of length 2 that contains p and n-p.

5.1.5 Akaike Information Criteria

The Akaike Information Criteria has an Expected value equal to the SSE/n in the validation data set for normal data. Hastie and Tibshirani call this the Predicted Square Error. It also is the expected Predicted Deviance for logistic regressions. This is easily computed, and can be used for both selecting degrees of freedom or model selection (hypothesis testing). It is particularly useful because the models do not have to be exactly nested.

aic(model.object, scale)
Calculates and prints the Akaike information criteria for any model along with the total and residual degrees of freedom and the residual deviance. The scale will be determined from the family of the model.object if the scale argument is omitted.

Note: we will define the `aic()` function during lab!

5.2 Laboratory for Week 5: Nonparametric smoothing example

5.2.1 Basic glm model for all cause mortality.

1. Recreate our basic glm mortality model. The OK variable is the same as in Exercise 1.

```
> OK_!is.na(age) & !is.na(bmi) & !is.na(ser.chol) & !is.na(bp1sys)
> OK_OK & !is.na(race) & !is.na(booze) & !is.na(sex) & !is.na(smokever)
```

or, you can try:

```
> OKvars_c("age","bmi","ser.chol","bp1sys","race","booze","sex","smokever")
> OK_makeOK(OKvars)
```

and then fit the model:

```
> mfit_glm(d.total~sex+race+factor(booze)+bp1sys+ns(age,4)+ser.chol,
+ family=binomial, subset=(OK))
> summary(mfit, c=F)
```

5.2.2 Defining the `aic()` function

We can construct an Splus function which computes Akaike's information criterion using components from the GAM fit. Recall equation 6.32 from Hastie & Tibshirani:

$$AIC = \frac{1}{n}\sum_{i=1}^{n} D(y_i;\hat{\mu}_i) + 2\mathrm{tr}(\mathbf{R})\phi/n$$

This translates into the following Splus code:

```
> aic <- function(fit)
+ {
+     n <- length(fit$fitted)
+     scale <- deviance.lm(fit)/fit$df.resid
+     deviance(fit) + 2 * (n - fit$df.resid) * scale
+ }
```

5.2.3 Parametric smoothing for serum cholesterol

Record model deviance and degrees of freedom for each model. Compare changes in model deviances between all models.

2. Quintile smoother.

```
> chol5_cut(ser.chol, quantile(ser.chol, c(0,.2,.4,.6,.8,1)), include.lowest=T)
> mfit.smu_update(mfit, .~.-ser.chol+factor(chol5))
> anova(mfit, mfit.smu, test="Chi")
> plot.gam(mfit.smu)
> aic(mfit.smu)
```

3. Polynomial smoother with a cubic polynomial.

```
> mfit.smu_update(mfit, . ~ . - ser.chol + poly(ser.chol, 3))
> anova(mfit, mfit.smu, test="Chi")
> plot.gam(mfit.smu)
> aic(mfit.smu)
```

4. Natural spline smoother.

```
> mmodel.smu_update(mfit, . ~ . - ser.chol + ns(ser.chol, 4))
> anova(mfit, mfit.smu, test="Chi")
> plot.gam(mfit.smu)
> aic(mfit.smu)
```

5.2.4 Nonparametric smoothing for serum cholesterol

5. Loess smoother. Note the nonlinear degrees of freedom for the smoother in the summary.

```
> mfit.smu_gam(d.total~sex+race+factor(booze)+bp1sys+ns(age,4)+lo(ser.chol,span
> summary(mfit.smu)
> anova(mfit, mfit.smu, test="Chi")
> plot(mfit.smu)
> aic(mfit.smu)
```

6. Add more degrees of freedom by lowering the span to 70serum cholesterol.

```
> mfit.smu2_update(mfit.smu, . ~ . - lo(ser.chol,span=0.75) + lo(ser.chol,span=
> summary(mfit.smu)
> anova(mfit.smu, mfit.smu2, test="Chi")
> aic(mfit.smu2)
```

5.2.5 Serum cholesterol and cause-specific mortality

7. Next do the same thing for ischemic heart disease (`d.heart`) deaths. Plot the adjusted smooth for serum cholesterol. What does the relationship look like?

```
> mfit.heart_update(mfit.smu, d.heart ~ . )
> summary(mfit.heart)
> plot(mfit.heart)
> aic(mfit.heart)
```

8. Finally, repeat for cancer deaths (d.cancer). Plot the adjusted smooth for serum cholesterol. Do you know have an idea why the shape of the all cause curve was the way it appeared?

```
> mfit.cancer_update(mfit.smu, d.cancer ~ . )
> summary(mfit.cancer)
> plot(mfit.cancer)
> aic(mfit.cancer)
```

5.3 Homework for Week 5

5.3.1 Effect of pre-clinical cancer on serum cholesterol

1. Some people have speculated that early stage cancer results in changes in cholesterol concentrations. We can exclude most people who had early stage cancer at examination by restricting the analysis to a subset where the difference between the age at enrollment (age) and the age at death (d.age) is greater than 5 years. But first we must assign a large value to d.age for individuals who have not died by the end of the study and thus have a missing value for d.age. What happens to the relationship between cancer mortality and serum cholesterol?

```
> hanes1a$d.age_ifelse(d.total==0,age+20,d.age)
> mfit.cancer_update(mfit.cancer, subset=( OK & I(hanes1a$d.age-age) > 5 ))
> summary(mfit.cancer)
> plot(mfit.cancer)
```

2. Some people have speculated that early stage cancer results in changes in Perhaps the pre-clinical period was too short. Exclude the seven years prior to death from the analysis, that is d.age - age ¡ 7. What happens to the relationship between cancer mortality and serum cholesterol?

```
> mfit.cancer_update(mfit.cancer, subset=( OK & I(hanes1a$d.age-age) > 7 ))
> summary(mfit.cancer)
> plot(mfit.cancer)
```

6 Week 6

6.1 Preparation for Week 5: Advanced Generalized Additive Models in SPLUS

6.1.1 Stepwise selection of GAM models

`step.gam(gam.object, scope=scope.list)`
determines the optimum model in terms of minimizing the AIC statistic from a range of models defined by the `scope` list using a step-wise search. The step-wise-selected model is returned along with an "anova" component corresponding to the steps taken in the search. The gam.object is used as the starting model and any terms in the formula of the gam.object that are not contained in any of the terms in the scope.list will be forced to be present in every model considered.

`scope.list`
a list of formulas, with each formula corresponding to a term in the model. A "1" in the formula allows the additional option of leaving the term out of the model entirely. Each of the formulas in scope specifies a "regimen" of candidate forms in which *a particular term* (i.e. for the term `age`, we might use the list element `~1+age+ns(age)+lo(age)` to try each of these terms for age in the model.) One variable from each of the terms in the scope.list must be present in the formula of the gam.object.

6.1.2 Generalized Estimating Equations (GEE)

`library(gee)`
loads the Splus language library with the `gee()` function. This must be run prior to any call to the `gee()` function.

`gee(formula, id, family, data, subset, na.action, corstr="independence", M=1)`
Produces an object of class `"gee"` which is a Generalized Estimation Equation fit of the data. The `formula`, `family`, `data`, `subset`, and `na.action` arguments are the same as in `glm()`. The `id` argument specifies a vector which identifies the clusters. The length of `id` should be the same as the number of observations (it is usually a variable which is to be extracted from your `data.frame`). Data are assumed to be sorted so that observations on a cluster are contiguous rows for all entities in the formula.

The working correlation structure is specified by a character string in the corstr argument. The following are permitted: `"independence"`, `"fixed"`, `"stat_M_dep"`,

`"non_stat_M_dep"`, `"exchangeable"`, `"AR-M"`, and `"unstructured"`. When the corstr argument is `"stat_M_dep"`, `"non_stat_M_dep"`, or `"AR-M"` then the `M` argument must be specified with an integer value.

6.1.3 Random Effects in Linear Models

The `lme()` function has lots of arguments! The particular arguments that we are concerned with are the ones listed below, however, please do look at the Splus help page on `lme` if you'd like to see them completely enumerated.

`fixed` a formula specifying the fixed effects part of the model. The dependent variable is always specified as a fixed effect. A model with the intercept as the only fixed effect can be specified as `y~1`.

`cluster` an expression (usually just the name of a variable in your data.frame, such as locode or id) specifying the experimental units over which the random effects vary.

`random` a formula specifying the random effects part of the model, but with no left side to the ~ in the formula expression. By default `random` is set equal to `fixed`. There need not be a random effect formula for each parameter. A model with the intercept as the only random effect can be specified as `~1`. If the intercept is not a random effect, the intercept must be specifically excluded with "`- 1`. Note: random effects are always assumed to have mean zero. A nonzero mean can be specified by including an identical term in the fixed effects part of the model.

`est.method` estimation method; if equal to `"ML"`, Maximum Likelihood is used, otherwise if `"RML"` is specified, Residual Maximum Likelihood (sometimes also called *Restricted* Maximum Likelihood) is used. Partial matching of arguments is used, so only the first character needs to be provided. Default is `"RML"`.

`re.block` a list indicating how the random effects should be blocked. Random effects pertaining to different blocks are assumed to be independent. The random effects can be referenced either by their names, or the order in which they appear in the random formula. By default all random effects are included in the same block.

`re.structure` a character vector describing the variance-covariance structure in each block of the random effects. Predefined structure names are listed in the following table, but users can define their own variance-covariance structure. If `re.structure` is of length 1, the name will be used for all blocks. Partial matching against the predefined names is used, so only the first character needs to be provided. Default is `"unstructured"`.

Variance-Covariance Matrix	String	Number of parameters
General	"unstructured"	q(q+1)/2
Independent random effects	"diagonal"	q
Constant variance	"identity"	1
Compound symmetry	"compsymm"	2
Autoregressive - 1st order	"ar1"	2

`serial.structure` the variance-covariance structure to be used for the within-cluster errors. Predefined structure names are listed in the table, but users can define their own serial structure named as a class with methods for the generic functions. Default is `"identity"`.

Variance-Covariance Matrix	String	Number of parameters
Constant variance	"identity"	1
Autoregressive - 1st order	"ar1"	2
Autoregressive - 2nd order	"ar2"	3
Moving average - 1st order	"ma1"	2
Moving average - 2nd order	"ma2"	3
Autoregressive moving average	"arma11"	3

`serial.covariate` a formula object specifying the time covariate to be used with serial.structure. This argument has the same characteristics as random. If serial.structure is not equal to `"identity"` and `noserial.covariate` is provided, the time covariate will be set to the order of the observations within each cluster.

6.1.4 Linear Mixed Effect Object

`lme.object$coefficients` a list with two components, fixed and random, where the first is a vector containing the estimated coefficients for the fixed effects and the second is a matrix containing the estimated coefficients for the random effects. The columns refer to the parameters in the random formula, and the rows to the cluster levels; the names of the coefficients are the same as those in the fixed or random formulas when just a . is used on the right hand side, or the original name with the covariate name appended to it (i.e. name.covariate) otherwise.

`lme.object$fitted.values` a data frame containing the population and cluster fitted values. The population fitted values are evaluated at the converged estimates of the fixed effects and the mean of the random effects (That is the random effects are set to zero). The cluster fitted values are evaluated at the converged estimates of the fixed effects and the conditional estimates of the random effects.

`lme.object$residuals` a data frame containing the population and cluster residuals from the fit. The population residuals are the observed values minus the population fitted values and the cluster residuals are the observed values minus the cluster fitted

values. If the variance function structure used in the fit is different than `"identity"`, a column with the estimated individual standard deviations will be included in the residuals data frame.

`lme.object$var.ran` Random effects variance-covariance-correlation matrix estimate. The variances estimates for the random effects are displayed on the main diagonal, covariances above the diagonal and correlations below the diagonal.

`lme.object$re.block` and `lme.object$re.structure` the blocking and covariance structures used for the random effects.

6.2 ComputingLab for Week 6: Generalized additive model example

6.2.1 Bivariate smoothing

1. GAM logistic regression model for all cause mortality among men with nonparametric loess smooths for age and serum cholesterol.

```
> mfit_gam(d.total~lo(age,span=0.6)+lo(ser.chol,span=0.7)+bmi,
+    family=binomial, data=hanes1a,subset=OK&sex==1)
> summary.glm(mfit,c=F,disp=0)
> plot(preplot(mfit)[[2]],se=T)
```

2. Now examine interaction between age and serum cholesterol using bivariate smoothing. We need to increase our maximum object size, since this is a big plot. (Some of you who have limited system resources may have trouble with this, so you'll have to skip this part.)

```
> options(object.size=10e+07)
> mfit.3d_update(mfit,.~lo(age,ser.chol,span=0.8))
> summary.glm(mfit.3d,c=F,disp=0)
> plot(mfit.3d)
```

   Now compare the univariate vs. bivariate models. Which gives a better fit?

```
> aic(mfit)
> aic(mfit.3d)
> anova(mfit,mfit.3d,test="Chi") # Note: are these nested?  No!
```

   Note that it is also possible to produce a surface plot of the simultaneously fitted univariate smooths, but it is a lot of complicated Splus code, so we're not going to do it in lab. If you're interested, I can provide you with code.

6.2.2 Using `step.gam` to find the best smoothing span

3. Recreate the GAM model for total mortality without any smoothed terms:

```
> mfit_gam(d.total~age+bmi+bp1sys+ser.chol,
+    family=binomial, subset=OK & sex==1)
> summary.glm(mfit,c=F,disp=0)
> aic(mfit)
```

4. Now use `step.gam()` to find the span for serum cholesterol that minimizes the AIC. What is the best span? Is this a maximum likelihood result?

```
> step.gam(mfit,scope=list(~age,~bmi,~bp1sys,
+     ~ser.chol+lo(ser.chol,0.7)+lo(ser.chol,0.75)+lo(ser.chol,0.8)
+     +lo(ser.chol,0.85)+lo(ser.chol,0.90)))
> step.gam(mfit,scope=list(~age,~bmi,~bp1sys,
+     ~ser.chol+lo(ser.chol,0.81)+lo(ser.chol,0.82)+lo(ser.chol,0.83)
+     +lo(ser.chol,0.84)+lo(ser.chol,0.85)))
```

5. Now find the optimal spans for age, body mass index (bmi), and blood pressure (bp1sys). Under what circumstances would the individual optimization of each span provide a different result from optimizing all of the spans jointly?

```
> step.gam(mfit,scope=list(
+     ~age + lo(age,0.80) + lo(age,0.85) + lo(age,0.90),
+     ~bmi + lo(bmi,0.80) + lo(bmi,0.85) + lo(bmi,0.90),
+     ~bp1sys + lo(bp1sys,0.80) + lo(bp1sys,0.85) + lo(bp1sys,0.90),
+     ~ser.chol + lo(ser.chol,0.80) + lo(ser.chol,0.85) + lo(ser.chol,0.90)))
```

6.2.3 Linear Models for Blood Pressure

6. Recreate our linear model for blood pressure and add serum calcium as an independent variable to the model. But first we must subset our data to complete information on serum calcium, (and only small city clusters, i.e., those having `locode<50` for compatibility with a later result) so we'll make (yet another) "`OK.`" variable:

```
> newOK_OK & !is.na(ser.calc) & locode<50
```

...and set up our model formula, (we can use the same model formula for many different modeling techniques)

```
> model_bp1sys~sex+as.factor(race)+smokever+as.factor(booze)+age+ser.calc
```

And now, fit our model:

```
> bp.fit_glm(model,data=hanes1a,subset=newOK)
> summary(bp.fit,c=F)
```

7. Now switch to a robust gaussian error distribution. Record the effect estimates ad standard errors for age and serum calcium.

```
summary(update(bp.fit,family=robust(gaussian)),c=F)
```

6.2.4 Generalized Estimating Equations for Blood Pressure

The NHANES I study was not a simple random sample of the US population, since providing standardized medical examinations would have been too costly. Rather a limited number of counties or county clusters were selected at the first stage and then subjects were selected within these locations. Persons from the same area are often more alike than two randomly selected persons reflecting the clustering of ethnic immigrants or racial minorities or the commonality of exposures and socioeconomic differences. If all the causes of these similarities are not included in the model, the errors will be correlated within the same location (block), but uncorrelated between locations. This is called a design effect and can be modeled using the `"exchangeable"` correlation structure.

8. Now use a Generalized Estimating Equation (GEE) to model the error structure. Since the gee() function is not part of standard Splus, you first must attach the gee library. The gee() function assumes that the data.frame has been sorted by the id variable. Record the effect estimates and both the naive and robust standard errors for age and serum calcium.

```
> library(gee)
> bp.fit.gee_update(bp.fit, class="gee", id=locode, corstr="exchangeable")
> summary(bp.fit.gee,c=F)
```

9. What might have happened if we had started from a model that assumed independent errors? Substitute "independence" for "exchangeable" in the preceding model. Record the effect estimates and both the naive and robust standard errors for age and serum calcium. How do these standard errors differ from those for the preceding model?

```
> summary(update(bp.fit.gee,corstr="independence"),c=F)
```

6.2.5 Random effects model for blood pressure

10. The effect of serum calcium on blood pressure may not be the same within all locations. We can model this by allowing location specific effect estimates (random effects) that vary around a common mean effect (fixed effect). We can fit both a fixed effect and a random effect for ser.calc across the various locations using the lme() function. Remember that an intercept is automatically included on the right-hand side of every formula expression unless the intercept is specifically excluded with a -1. Compare the fixed effect in this model to the effects in previous models and examine the distribution of the random effects.

```
> bp.fit.lme_lme(fixed=model,random=~ser.calc-1,cluster=~locode,
+    data=hanes1a,subset=newOK)
> summary(bp.fit.lme,re=F)
> summary(bp.fit.lme$coefficients$random)
```

6.2.6 Generalized Linear Models for Mortality

11. Well if serum calcium is related to increased blood pressure, is it also related to mortality? Recreate our logistic model for all-cause mortality and add serum calcium to that model. Record the effect estimate and standard error for serum calcium.

```
> model_d.total~sex+as.factor(race)+smokever+as.factor(booze)+age+ser.calc
> d.fit_glm(model, family=binomial, data=hanes1a, subset=newOK)
> summary(d.fit,c=F,disp=0)
```

Now switch to a robust binomial error distribution. Record the effect estimate and standard error for serum cholesterol.

```
> summary(update(d.fit,family=robust(binomial)),c=F,disp=0)
```

6.2.7 Generalized Estimating Equation for Mortality

12. Now use a Generalized Estimating Equation (GEE) with an "exchangeable" correlation structure to model the error structure. Record the effect estimate and both the naive and robust standard errors for serum calcium.

```
> d.fit.gee_update(d.fit, class="gee", id=locode, corstr="exchangeable")
> summary(d.fit.gee,c=F)
```

7 Week 7

7.1 Preparation for Week 7: Time Series Analysis in Splus

7.1.1 Philadelphia mortality time series data set

date A SAS date variable: the number of days since January 1, 1960.

total today's number of deaths

winter a binary indicator for Winter (Dec, Jan, Feb)

spring a binary indicator for Spring (Mar, Apr, May)

summer a binary indicator for Summer (Jun, Jul, Aug)

tpm today's average total suspended particulates (TSP)

tpm1 yesterday's average total suspended particulates (TSP)

tpma two day moving average of total suspended particulates (TSP)

temp today's minimum temperature in degrees F

dew today's average dew point temperature

hot a binary indicator of 159 days whose minimum temperature exceeded 80 degrees F

humid a binary indicator of 346 days whose dew point temperature exceeded 67 degrees F

7.1.2 Time series functions

`month.day.year(Julian.date, origin=c(month = mm, day = dd, year=yyyy))`
returns a list with three named numeric vectors: month, day, and year. Julian.date is a numeric integer vetor of Julian Dates, that is the number of days since origin. Origin is a vector specifying the origin as month, day, and year. If missing, it defaults to options(chron.origin) if this is non-null, otherwise c(month=1, day=1, year=1960).

`day.of.week(month, day, year)`
returns the day of week (0 to 6): 0 refers to Sunday.

`ts(data=SAS.date, start=1960, frequency=365`
converts a SAS date variable into an Splus time series object.

`tsp(x)`

returns the tsp attribute of x: starting time, ending time, and the observation frequency of the time series. The argument x may be any object, but the returning value will be NULL if x is not a time series. Missing values (NAs) are allowed.

7.2 ComputingLab for Week 7: Time series decomposition

7.2.1 Preliminaries

The source code[8], output[9], and graphs[10] for this lab are available on the webpage.

1. Get new data.frame (philly) from a file (philly.dmp) in ~eeb271/PUBLIC.

```
> data.restore("/usr1/users/biostat/eeb271/datasets/philly.dmp")
```

2. Create a day-of-week variable using Splus date functions. Renumber dates from the beginning of the study. Then mount the data.frame and examine its contents.

```
> summary(philly$date)
> month.day.year(4749)
> new.date_month.day.year(philly$date)
> names(new.date)
> philly$dow_day.of.week(new.date$month,new.date$day,new.date$year)
> philly$dow[1:10]
> philly$date_philly$date-4748  #date will be # days since 1/1/73
                                # Be careful!  only do this 1 time!
```

3. Now, attach the frame and increase your object size, because time-series models need a lot of memory!

```
> attach(philly)
> objects(2)
> options(object.size=10E8)
```

4. Examine the daily count of mortality with respect to a Poisson process. Which transformation (log or sqrt) is better for a Poisson distribution in terms of variance stablization? Note: qqnorm does a QQ plot of the variable given as an argument. Here, we want to put 4 plots on 1 page.

```
> summary(total)
> par(mfrow=c(2,2))
> qqnorm(total)
> qqnorm(log(total))
> qqnorm(sqrt(total))
> par(mfrow=c(1,1)) # set this back to 1 plot per page
```

[8] http://biosun1.harvard.edu/ eeb271/lab7.q
[9] http://biosun1.harvard.edu/ eeb271/lab7.out
[10] http://biosun1.harvard.edu/ eeb271/lab7.ps

7.2.2 Making titles for GAM plots

Throughout the rest of this lab, we are going to be making several GAM plots. I find it useful in keeping track of things to provide a title for the plots which displays the terms used to fit the model. Let's make a function to do this:

```
> gamtitle_function(fit)
+    title(paste(names(fit$coefficients)[-1],collapse="+"))
```

7.2.3 Trend and season

5. Compare a Poisson model for total daily mortality with linear trend to a model with a loess smooth with a large span. Is there a time-trend? Is it linear?

```
> philly.glm_glm(total~poly(date,2),
+    family=poisson, data=philly, na.action=na.omit)
> summary.glm(philly.glm, c=F, disp=0)
>
> philly.gam_update(philly.glm, .~lo(date,0.7),class="gam")
> summary(philly.gam)
> summary.glm(philly.gam,c=F,disp=0)
> plot(philly.gam,se=T)
> gamtitle(philly.gam)
>
> aic(philly.glm)
> aic(philly.gam)
```

6. Extract seasonal pattern using a loess smooth with a small span (120/2921 = 0.04). Is the seasonal pattern constant? What variations do you see? What could cause them? Could the observed seasonal pattern have been easily fit by a parametric function?

```
> philly.glm_update(philly.glm, .~.+winter+spring+summer)
> summary.glm(philly.glm,c=F,disp=0)
>
> philly.gam_update(philly.gam, .~lo(date,0.04))
> summary(philly.gam)
> plot(philly.gam,se=T)
> mytitle(philly.gam)
>
> aic(philly.gam)
> aic(philly.glm)
```

7. Extract both long-term trend and seasonal pattern simultaneously. How many total degrees of freedom have been used to model trend and season? Would an equivalent number of seasonal dummy variables have done as well? What is the real width of the seasonal span in terms of total days, that is the number of days with relatively high weights in the smooth?

```
> philly.gam_update(philly.gam,.~lo(date,0.7)+lo(date,0.04))
> summary(philly.gam)
> plot(philly.gam,se=T)
> # how does this compare to using a poly term for the long term trend?
> aic(philly.gam)
> aic(update(philly.gam,.~poly(date,2)+lo(date,0.04)))
```

7.2.4 Air pollution

8. Examine models with today's total suspended particles (tpm) or yesterday's total suspended particles (tpm1). Carefully examine the anova comparison of the two models. Is the effect of particulate matter delayed or immediate? What about the degrees of freedom? How correlated are these two measures of particulate matter?

```
> philly.gam_update(philly.gam, .~.+tpm)
> summary.glm(philly.gam,c=F,disp=0)
> philly.gam2_update(philly.gam, .~.+tpm1)
> summary.glm(philly.gam2,disp=0) # look at the correlation this time!
> anova(philly.gam,philly.gam2,test="Chi")
```

9. Use two-day average of total suspended particulates (tpma)

```
> philly.gam_update(philly.gam, .~.-tpm-tpm1+tpma)
> summary.glm(philly.gam, c=F,disp=0)
```

10. Create a nonlinear model for total suspended particulates (tpma). Explore the effect of reasonable changes in the span for tpma.

```
> # note the order of the updated formula here:
> philly.gam2_update(philly.gam, .~lo(tpma, 0.75)+.-tpma)
> summary.glm(philly.gam2, disp=0)
> summary(philly.gam2)
> plot(preplot(philly.gam2)[[1]],se=T)
> mytitle(philly.gam2)
```

7.2.5 Weather and day of week

11. Examine temperature as a confounder of a linear particle effect. What does this function look like? Could it have been easily fit by a parametric function? Does temperature confound? Was a linear adjustment of temperature sufficient to control for confounding?

```
> # first, look at the linear effect of temp:
> summary.glm(philly.glm_update(philly.glm, .~.+temp), disp=0, c=F)
>
> philly.gam_update(philly.gam, .~lo(temp,0.75)+.)
> summary(philly.gam)
> plot(preplot(philly.gam)[[1]],se=T)
> mytitle(philly.gam)
> summary.glm(philly.gam,c=F,disp=0)
```

12. Add prior day's temperatures (templ) to the model. What happens to the coefficient for tpma? What happens to the coefficient for current day's temperature (temp)? Why?

```
> summary.glm(update(philly.glm, .~.+templ),c=F,disp=0)
```

13. Examine dew point temperature as a confounder of a linear particle effect. What does this function look like? Could it have been easily fit by a parametric function? Does dew point temperature confound? Was a linear adjustment for dew point temperature sufficient to control for confounding?

```
> summary.glm(philly.glm_update(philly.glm, .~.+dew), disp=0, c=F)
>
> summary(philly.gam_update(philly.gam,.~lo(dew,0.75)+.))
> summary.glm(philly.gam,c=F,disp=0)
> plot(preplot(philly.gam)[[1]],se=T)
> mytitle(philly.gam)
```

14. Examine day of the week as a confounder of a linear particle effect. What does this functio look like? Could it have been easily fit by a parametric function? Does day of week confound? Does a loess smooth through day of week sufficiently control confounding with fewer degrees of freedom than a separate indicator for each day?

```
> summary.glm(philly.glm_update(philly.glm, .~.+as.factor(dow)),c=F,disp=0)
>
> summary(philly.gam_update(philly.gam, .~lo(dow,0.75)+.))
> plot(preplot(philly.gam)[[1]],se=T)
> mytitle(philly.gam)
> summary.glm(philly.gam,c=F,disp=0)
```

8 EEB271c Final Exam, Spring 1999

Everything you need for the final exam is located in directory `eeb271/final`[11] on hsph. In particular, you should look at the file called `README` which explains what everything else is in the directory.

You are expected to work on the exam independently, and turn in your completed solutions on Friday, March 26 to Joel, Louise, or Carrie.

Please email Louise (`ryan@jimmy.harvard.edu`) and/or Carrie (`cwager@hsph.harvard.edu`) if you have any questions or computing problems.

[11] http://biosun1.harvard.edu/ eeb271/final

Network Working Group
RFC # 610
NIC # 21352

Richard Winter, Jeffrey Hill, Warren Greiff
CCA

December 15, 1973

Further Datalanguage Design Concepts

Richard Winter
Jeffrey Hill
Warren Greiff

Computer Corporation of America
December 15, 1973

Acknowledgment

During the course of the Datacomputer Project, many people have contributed to the development of datalanguage.

The suggestions and criticisms of Dr. Gordon Everest (University of Minnesota), Dr. Robert Taylor (University of Massachusetts), Professor Thomas Cheatham (Harvard University) and Professor George Mealy (Harvard University) have been particularly useful.

Within CCA, several people in addition to the authors have participated in the language design at various stages of the project. Hal Murray, Bill Bush, David Shipman and Dale Stern have been especially helpful.

1. Introduction

1.1 The Datacomputer System

The datacomputer is a large-scale data utility system, offering data storage and data management services to other computers.

The datacomputer differs from traditional data management systems in several ways.

First, it is implemented on dedicated hardware, and comprises a separate computing system specialized for data management.

Second, the system is implemented on a large scale. Data is intended to be stored on mass storage devices, with capacities in the range of a trillion bits. Files on the order of one hundred billion bits are to be kept online.

Third, it is intended to support sharing of data among processes operating in diverse environments. That is, the programs which share a given data base may be written in different languages, execute on different hardware under different operating systems, and support end users with radically different requirements. To enable such shared use of a data base, transformations between various hardware representations and data structuring concepts must be achieved.

Finally, the datacomputer is designed to function smoothly as a component of a much larger system: a computer network. In a computer network, the datacomputer is a node specialized for data management, and acting as a data utility for the other nodes. The Arpanet, for which the datacomputer is being developed, is an international network which has over 60 nodes. Of these, some are presently specialized for terminal handling, others are specialized for computation (e.g., the ILLIAC IV), some are general purpose service nodes (e.g., MULTICS) and one (CCA) is specialized for data management.

1.2 Datalanguage

Datalanguage is the language in which all requests to the datacomputer are stated. It includes facilities for data description and creation, for retrieval of or changes to stored data, and for access to a variety of auxiliary facilities and services. In datalanguage it is possible to specify any operation the datacomputer is capable of performing. Datalanguage is the only language accepted by the datacomputer and is the exclusive means of access to data and services.

1.3 Present Design Effort

We are now engaged in developing complete specifications for datalanguage; this is the second iteration in the language design process.

A smaller, initial design effort developed some concepts and principles which are described in the third working paper in this series. These have been used as the basis of software implementations resulting in an initial network service capability. A user manual for this system was published as working paper number 7.

As a result of experience gained in implementation and service, through further study of user requirements and work with potential users, and through investigation of other work in the data management field, quite a few ideas have been developed for the improvement of datalanguage. These are being assimilated into the language design in the iteration now in progress.

When the language design is complete, it will be incorporated into the existing software (requiring changes to the language compiler, but having little impact on the rest of the system).

Datacomputer users will first have access to the new language during 1975.

1.4 Purpose of this Paper

This paper presents concepts and preliminary results, rather than a completed design. There are two reasons for publishing now.

The first is to provide information to those planning to use the datacomputer. They may benefit from knowledge of our intentions for development.

The second is to enable system and language designers to comment on our work before the design is frozen.

1.5 Organization of the Paper

The remainder of the paper is divided into four sections.

Section 2 discusses the most global considerations for language design. This comprises our view of the problem; it has influenced our work to date and will determine most of our actions in completion of the design. This section provides background for section 3, and reviews some

material that will be familiar to those who have been following our work closely.

Section 3 discusses some of the specific issues we have worked on. The emphasis is on solutions and options for solution.

In sections 2 and 3 we are presenting our "top-down" work: this is the thinking we have done based on known requirements and our conception of the desirable properties of datalanguage.

We have also been working from the opposite end, developing the primitives from which to construct the language. Section 4 presents our work in this area: a model datacomputer which will ultimately provide a precise semantic definition of datalanguage. Section 4 explains that part of the model which is complete, and relates this to our other work.

Section 5 discusses work that remains, both on the model and in our top-down analysis.

2. Considerations for Language Design

2.1 Introduction

Data management is the task of managing data as a resource, independent of hardware and applications programs. It can be divided it into five major sub-tasks:

(1) _creating_ databases in storage,
(2) making the data _available_ (e.g., satisfying queries),
(3) _maintaining_ the data as information is added, deleted and modified,
(4) assuring the _integrity_ of the data (e.g., through backup and recovery systems, through internal consistency checks),
(5) _regulating_access_, to protect the databases, the system, and the privacy of users.

These are the major data-related functions of the datacomputer; while the system will ultimately provide other services (such as accounting for use, monitoring performance) these are really auxiliary and common to all service facilities.

This section presents global considerations for the design of datalanguage, based on our observations about the problem and the environment in which it is to be solved. The central problem is data management, and the datacomputer shares the same goals as many currently available data management systems. Several aspects of the datacomputer create a unique set of problems to be solved.

2.2 Hardware Considerations

2.2.1 Separate Box

The datacomputer is a complete data management utility in a separate, closed box. That is, the hardware, the data and the data management software are segregated from any general-purpose processing facilities. There is a separate installation dedicated to data management. Datalanguage is the only means users have for communicating with the datacomputer and the sole activity of the datacomputer is to process datalanguage requests.

Dedicating hardware provides an obvious advantage: one can specialize it for data management. The processor(s) can be modified to have data management "instructions"; common low-level software functions can be built into the hardware.

A less obvious, but possibly more significant, advantage is gained from the separateness itself. The system can be more easily protected. A fully-developed datacomputer on which there is only maintenance activity can provide a very carefully controlled environment. First, it can be made as physically secure as required. Second, it needs to execute only system software developed at CCA; all user programs are in a high-level language (datalanguage) which is effectively interpreted by the system. Hence, only datacomputer system software processes the data, and the system is not very vulnerable to capture by a hostile program. Thus, since there is the potential to develop data privacy and integrity services that are not available on general-purpose systems, one can expect less difficulty in developing privacy controls (including physical ones) for the datacomputer than for the systems it serves.

2.2.2 Mass Storage Hardware

The datacomputer will store most of its data on mass storage devices, which have distinctive access characteristics. Two examples of such hardware are Precision Instruments' Unicon 690 and Ampex Corporation's TBM system. They are quite different from disks, and differ significantly from one another.

However, almost all users will be ignorant of the characteristics of these devices; many will not even know that the data they use is at the datacomputer. Finally, as the development of the system progresses, data may be invisibly shunted from one datacomputer to another, and as a result be stored in a physical format quite different from that originally used.

In such an environment, it is clear that requests for data should be stated in logical, not physical terms.

2.3 Network Environment

The network environment provides additional requirements for datacomputer design.

2.3.1 Remote Use

Since the datacomputer is to be accessed remotely, the requirement for effective data selection techniques and good mechanisms for the expression of selection criteria is amplified. This is because of the narrow path through which network users communicate with the datacomputer. Presently, a typical process-to-process transfer rate over the Arpanet is 30 kilobits per second. While this can be increased through optimization of software and protocols, and through additional

expenditure for hardware and communications lines, it seems safe to assume that it will not soon approach local transfer rates (measured in the megabits per second).

A typical request calls for either transfer of part of a file to a remote site, or for selective update to a file already stored at the datacomputer. In both of these situations, good mechanisms for specifying the parts of the data to be transmitted or changed will reduce the amount of data ordinarily transferred. This is extremely important because with the low per bit cost of storing data at the datacomputer, transmission costs will be a significant part of the total cost of datacomputer usage.

2.3.2 Interprocess Use of the Datacomputer System

Effective use of the network requires that groups of processes, remote from one another, be capable of cooperating to accomplish a given task or provide a given service. For example, to solve a given problem which involves array manipulation, data retrieval, interaction with a user at a terminal, and the generalized services of a language like PL/I, it may be most economical to have four cooperating processes. One of these could execute at the ILLIAC IV, one at the datacomputer, one at MULTICS, and one at a TIP. While there is overhead in setting up these four processes and in having them communicate, each is doing its job on a system specialized for that job. In many cases, the result of using the specialized system is a gain of several orders of magnitude in economy or efficiency (for example, online storage at the datacomputer has a capital cost two orders of magnitude lower than online costs on conventional systems). As a result, there is considerable incentive to consider solutions involving cooperating processes on specialized systems.

To summarize: the datacomputer must be prepared to function as a component of small networks of specialized processes, in order that it can be used effectively in a network in which there are many specialized nodes.

2.3.3 Common Network Data Handling

A large network can support enough data management hardware to construct more than one datacomputer. While this hardware can be combined into one even larger datacomputer, there are advantages to configuring it as two (or possibly more) systems. Each system should be large enough to obtain economies of scale in data storage and to support the data management software. Important data bases can be duplicated, with a copy at each datacomputer; if one datacomputer fails, or is cut off by

network failure, the data is still available. Even if duplicating the file is not warranted, the description can be kept at the different datacomputers so that applications which need to store data constantly can be guaranteed that at least one datacomputer is available to receive input.

These kinds of failure protection involve cooperation between a pair of datacomputers; in some sense, they require that the two datacomputers function as a single system. Given a system of datacomputers (which one can think of as a small network of datacomputers), it is obviously possible to experiment with providing additional services on the datacomputer-network level. For example, all requests could be addressed simply to the datacomputer-network; the datacomputer-network could then determine where each referenced file was stored (i.e., which datacomputer), and how best to satisfy the request.

Here, two kinds of cooperation in the network environment have been mentioned: cooperation among processes to solve a given problem, and cooperation among datacomputers to provide global optimizations in the network-level data handling problem. These are only two examples, especially interesting because they can be implemented in the near term. In the network, much more general kinds of cooperation are possible, if a little farther in the future. For example, eventually, one might want the datacomputer(s) to be part of a network-wide data management system, in which data, directories, services, and hardware were generally distributed about the network. The entire system could function as a whole under the right circumstances. Most requests would use the data and services of only a few nodes. Within this network-wide system, there would be more than one data management system, but all systems would be interfaced through a common language. Because the datacomputers represent the largest data management resource in the network, they would certainly play an important role in any network-wide system. The language of the datacomputer (datalanguage) is certainly a convenient choice for the common language of such a system.

Thus a final, albeit futuristic, requirement imposed by the network on the design of the datacomputer system, is that it be a suitable major component for network-wide data management systems. If feasible, one would like datalanguage to be a suitable candidate for the common language of a network-wide group of cooperating data management systems.

2.4 Different Modes of Datacomputer Usage

Within this network environment, the datacomputer will play several roles. In this section four such roles are described. Each of them imposes constraints on the design of datalanguage. We can analyze them in terms of four overlapping advantages which the datacomputer provides:

1. Generalized data management services
2. Large file handling
3. Shared access
4. Economic volume storage

Of course, the primary reason for using the datacomputer will be the data management services which it provides. However, for some applications size will be the dominating factor in that the datacomputer will provide for online access to files which are so large that previously only offline storage and processing were possible. The ability to share data between different network sites with widely different hardware is another feature provided only by the datacomputer. Economies of scale make the datacomputer a viable substitute for tapes in such applications as operating system backup.

Naturally, a combination of the above factors will be at work in most datacomputer applications. The following subsections describe some possible modes of interaction with the datacomputer.

2.4.1 Support of Large Shared Databases

This is the most significant application of the datacomputer, in nearly every sense.

Projects are already underway which will put databases of over one hundred billion bits online on the Arpanet datacomputer. Among these are a database which will ultimately include 10 years of weather observations from 5000 weather stations located all over the world. As online databases, these are unprecedented in size. They will be of international interest and be shared by users operating on a wide variety of hardware and in a wide variety of languages.

Because these databases are online in an international network, and because they are expected to be of considerable interest to researchers in the related fields, it seems obvious that there will be extremely broad patterns of use. A strong requirement, then, is a flexible and general approach to handling them. This requirement of providing different users of a database with different views of the data is an overriding concern of the datalanguage design effort. It is discussed separately in Section 2.5.

2.4.2 Extensions of Local Data management Systems

We imagine local data handling systems (data management systems, applications-oriented packages, text-handling systems, etc.) wanting to take advantage of the datacomputer. They may do so because of the

economics of storage, because of the data management services, or because they want to take advantage of data already stored at the datacomputer. In any case, such systems have some distinctive properties as datacomputer users: (1) most would use local data as well as datacomputer data, (2) many would be concerned with the translation of local requests into datalanguage.

For example, a system which does simple data retrieval and statistical analysis for non-programming social scientists might want to use a census database stored at the datacomputer. Such a system may perform a range of data retrieval functions, and may need sophisticated interaction with the datacomputer. Its usage patterns would make quite a contrast with those of a single application program whose sole use of the datacomputer involves printing a specific report based on a single known file.

This social-science system would also use some local databases, which it keeps at its own site because they are small and more efficiently accessed locally. One would like it to be convenient to think of data the same way, whether it is stored locally or at the datacomputer. Certainly at the lower levels of the local software, there will have to be differences in interfacing; it would be nice, however, if local concepts and operations could easily be translated into datalanguage.

2.4.3 File Level Use of the Datacomputer

In this mode of use, other computer systems take advantage of the online storage capacity of the datacomputer. To these systems, datacomputer storage represents a new class of storage: cheaper and safer than tape, nearly as accessible as local disk. Perhaps they even automatically move files between local online storage and the datacomputer, giving users the impression that everything is stored locally online.

The distinctive feature of this mode of use is that the operations are on whole files.

A system operating in this mode uses only the ability to store, retrieve, append, rename, do directory listings and the like. An obvious way to make such file level handling easily available to the network community is to make use of the File Transfer Protocol (see Network Information Center document #17759 -- File Transfer Protocol) already in use for host to host file transfer.

Although such "whole file" usage of the datacomputer would be motivated primarily by economic advantages of scale, data sharing at the file level could also be a concern. For example, the source files of common network software might reside at the datacomputer. These files have

little or no structure, but their common use dictates that they be available in a common, always accessible place. It is taking advantage of the economics of the datacomputer, more than anything else, since most of these services are available on any file system.

This mode of use is mentioned here because it may account for a large percentage of datalanguage requests. It requires only capabilities which would be present in datalanguage in any case; the only special requirement is to make sure it is easy and simple to accomplish these tasks.

2.4.4 Use of Datacomputer for File Archiving

This is another economics-oriented application. The basic idea is to store on the datacomputer everything that you intend to read rarely, if ever. This could include backup files, audit trails, and the like.

An interesting idea related to archiving is incremental archiving. A typical practice, with regard to backing up data stored online in a time-sharing system, is to write out all the pages which are different than they were in the last dump. It is then possible to recover by restoring the last full dump, and then restoring all incremental dumps up to the version desired. This system offers a lower cost for dumping and storage, and a higher cost for recovery; it is appropriate when the probability of needing a recovery is low. Datalanguage, then, should be designed to permit convenient incremental archiving.

As in the case of the previous application (file system), archiving is important as a design consideration because of its expected frequency and economics, not because it necessarily requires any extra generality at the language level. It may dictate that specialized mechanisms for archiving be built into the system.

2.5 Data Sharing

Controlled sharing of data is a central concern of the project. Three major sub-problems in data sharing are: (1) concurrent use, (2) independent concepts of the same database, and (3) varying representations of the same database.

Concurrent use of a resource by multiple independent processes is commonly implemented for data on the file level in systems in which files are regarded as disjoint, unrelated objects. It is sometimes implemented on the page level.

Considerable work on this problem has already been done within the

datacomputer project. When this work is complete, it will have some impact on the language design; by and large however, we do not consider this aspect of concurrent use to be a language problem.

Other aspects of the concurrent use problem, however, may require more conscious participation by the user. They relate to the semantics of collections of data objects, when such collections span the boundaries of files known to the internal operating system. Here the question of what constitutes an update conflict is more complex. Related questions arise in backup and recovery. If two files are related, then perhaps it is meaningless to recover an earlier state of one without recovering the corresponding state of the other. These problems are yet to be investigated.

Another problem in data sharing is that not all users of a database should have the same concept of that database. Examples: (1) for privacy reasons, some users should be aware of only part of the database (e.g., scientists doing statistical studies on medical files do not need access to name and address), (2) for program-data independence, payroll programs should access only data of concern in writing paychecks, even though skill inventories may be stored in the same database, (3) for global control of efficiency, simplicity in application programming, and program-data independence each application program should "see" a data organization that is best for its job.

To further analyze example (3), consider a database which contains information about students, teachers, subjects and also indicates which students have which teachers for which subjects. Depending on the problem to be solved, an application program may have a strong requirement for one of the following organizations:

(1) entries of the form (student,teacher,subject) with no concern about redundancy. In this organization an object of any of the three types may occur many times.

(2) entries of the form

```
            (student,       (teacher,subject),
                            (teacher,subject),
                            .
                            .
                            .
                            (teacher,subject))
```

(3) entries of the form

```
            (teacher,       subject,(student...student),
                            subject,(student...student),
                            subject,(student.. .student))
```

and other organizations are certainly possible.

One approach to this problem is to choose an organization for stored data, and then have application programs write requests which organize

output in the form they want. The application programmer applies his ingenuity in stating the request so that the process of reorganization is combined with the process of retrieval, and the result is relatively efficient. There are important, practical situations in which this approach is adequate; in fact there are situations in which it is desirable. In particular, if efficiency or cost is an overriding consideration, it may be necessary for every application programmer to be aware of all the data access and organization factors. This may be the case for a massive file, in which each retrieval must be tuned to the access strategy and organization; any other mode of operation would result in unacceptable costs or response times.

However, dependence between application programs and data organization or access strategy is not a good policy in general. In a widely-shared database, it can mean enormous cost in the event of database reorganization, changes to access software, or even changes in the storage medium. Such a change may require reprogramming in hundreds of application programs distributed throughout the network.

As a result, we see a need for a language which supports a spectrum of operating modes, including: (1) application program is completely independent of storage structure, access technique, and reorganization strategy, (2) application program parametrically controls these, (3) application program entirely controls them. For a widely-shared database, mode (1) would be the preferred policy, except when (a) the application programmer could do a better job than the system in making decisions, and (b) the need for this increment of efficiency outweighed the benefits of program-data independence.

In evaluating this question for a particular application, it is important to realize the role of global efficiency analysis. When there are many users of a database, in some sense the best mode of operation is that which minimizes the total cost of processing all requests and the total cost of storing the data. When applications come and go, as real-world needs change, then the advantages of centralized control are more likely to outweigh the advantages of optimization for a particular application program.

The third major sub-problem arises in connection with item level representations. Because of the environment in which it executes, each application program has a preferred set of formatting concepts, length indicators, padding and alignment conventions, word sizes, character representations, and so on. Once again it is better policy for the application program to be concerned only with the representations it wants and not with the stored data representation. However, there will be cases in which efficiency for a given request overrides all other factors.

At this level of representation, there is at least one additional consideration: potential loss of information when conversion takes place. Whoever initiates a type conversion (and this will sometimes be the datacomputer and sometimes the application program) must also be responsible for seeing that the intent of the request is preserved. Since the datacomputer must always be responsible for the consistency and the meaning of a shared database, there are some conflicts to be resolved here.

To summarize, it seems that the result of wide sharing of databases is that a larger system must be considered in choosing a data management policy for a particular database. This larger system, in the case of the datacomputer, consists of a network of geographically distributed applications programs, a centralized database, and a centralized data management system. The requirement for datalanguage is to provide flexibility in the management of this larger system. In particular, it must be possible to control when and where conversions, data re-organizations, and access strategies are made.

2.6 Need for High Level Communication

All of the above considerations point to the need for high level communication between the datacomputer and its users. The complex and distinct nature of datacomputer hardware make it imperative that requests be put to the datacomputer so that it can make major decisions regarding the access strategies to be used. At the same time, the large amounts of data stored and the demand of some users for extremely high transmission bandwidths make it necessary to provide for user control of some storage and transmission schemes. The fact that databases will be used by applications which desire different views of the same data and with different constraints means that the datacomputer must be capable of mapping one users request onto another users data. Interprocess use of the datacomputer means that datasharing must be completely controllable to avoid the need for human intervention. Extensive facilities for ensuring data integrity and controlling access must be provided.

2.6.1 Data Description

Basic to all these needs is the requirement that the data stored at the datacomputer be completely described in both functional and physical parameters. A high level description of the data is especially important to provide the sharing and control of data. The datacomputer must be able to map between different hardware and different applications. In its most trivial form this means being able to convert between floating point number representations on different machines. On

the other extreme it means being able to provide matrix data for the ILLIAC IV as well as being able to provide answers to queries from a natural language program, both addressed to the same weather data base. Data descriptions must provide the ability to specify the bit level representations and the logical properties and relationships of data.

2.6.2 Data integrity and Access Control

In the environment we have been describing, the problems of maintaining data integrity and controlling use of data assume extreme importance. Shared use of datacomputer files depends on the ability of the datacomputer to guarantee that the restrictions on data-access are strictly enforced. Since different users will have different descriptions, the access control mechanism must be associated with the descriptions themselves. One can control access to data by controlling access to its various descriptors. A user can be constrained to access a given data base only through one specific description which limits the data he can access. In a system where the updaters of a database may be unknown to each other, and possibly have different views of the data, only the datacomputer can assure data integrity. For this reason, all restrictions on possible values of data objects, and on possible or necessary relationships between objects must be stated in the data description.

2.6.3 Optimization

The decisions regarding data access strategy must ordinarily be made at the datacomputer, where knowledge of the physical considerations is available. These decisions cannot be made intelligently unless the requests for data access are made at a high level.

For example, compare the following two situations: (1) a request calls for output of _all_ weather observations made in California exhibiting certain wind and pressure conditions, (2) a series of requests is sent, each one retrieving California weather observations; when a request finds an observation with the required wind and pressure conditions, it transmits this observation to a remote system. Both sessions achieve the same result: the transmission of a certain set of observations to a remote site for processing. In the first session, however, the datacomputer receives, at the outset, a description of the data that is needed; in the second, it processes a series of requests, each one of which is a surprise.

In the first case, a smart datacomputer has the option of retrieving all of the needed data in one access to the mass storage device. It can then buffer this data on disk until the user is ready to accept it. In

the second case, the datacomputer lacks the information it needs to make such an optimization.

The language should permit and encourage users to provide the information needed to do optimization. The cost of not doing it is much higher with mass storage devices and large files than it is in conventional systems.

2.7 Application Oriented Concerns

In the above sections we have described a number of features which the datacomputer system must provide. In this section we focus on what is necessary to make these features readily available to users of the datacomputer.

2.7.1 Datacomputer-user Interaction

An application interacts with the datacomputer in a _session_. A session consists of a series of requests. Each session involves connecting to the datacomputer via the network, establishing identities, and setting up transmission paths for both data and datalanguage. Datalanguage is transmitted in character mode (using network standard ASCII) over the datalanguage connection. Error and status messages are sent over this connection to the application program.

The data connection (called a PORT) is viewed as a bit stream and is given its own description. These descriptions are similar to those given for stored data. At a minimum this description must contain enough information for the datacomputer to parse the incoming bit stream. It also may contain data validation information as well. To store data at the datacomputer, the stored data must also have a description. The user supplies the mapping between the descriptions of the stored and transmitted data.

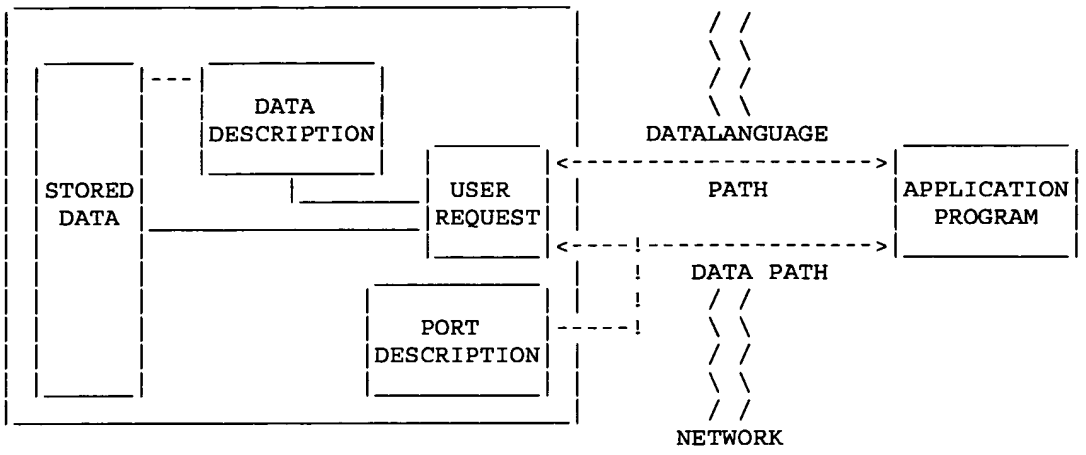


Figure 2-1
A Model of Datacomputer/User Interaction

2.7.2 Application Features for Data Sharing

In using data stored at the datacomputer, users may supply a description of the data which is customized to the application. This description is mapped onto the description of the stored data. These descriptions may be at different levels. That is, one may merely rearrange the order of certain items, while another could call for a total restructuring of the stored representation. So that each user may be able to build upon the descriptions of another, data entities should be given named types. These type definitions are of course to be stored along with the data they describe. In addition, certain functions are so closely tied to the data (in fact may be the data in the virtual description case -- see section 3), that they must also reside in the datacomputer and their tie with the data items should be maintained by the datacomputer. For example, one user can describe a data base as made up of structures containing data of the types _latitude_ and _longitude_. He could also describe functions for comparing data of this type. Other users, not concerned with the structure of the _latitude_ component itself, but interested in using this information simply to extract other fields of interest can then use the commonly provided definitions and functions. Furthermore, by adopting this strategy as many users as possible can be made insensitive to changes in the file which are tangential to their main interests. For example, _latitudes_ could be changed from binary representation to a character form and if use of that field were restricted to its definitions and associated functions, existing

application systems would be unaffected. Conversion functions could be defined to eliminate the impact on currently operating programs. The ability of such definitional facilities means that groups of users can develop common functions and descriptions for dealing with shared data and that conventions for use of shared data can be enforced by the datacomputer. These facilities are discussed under _extensibility_ in Section 3.

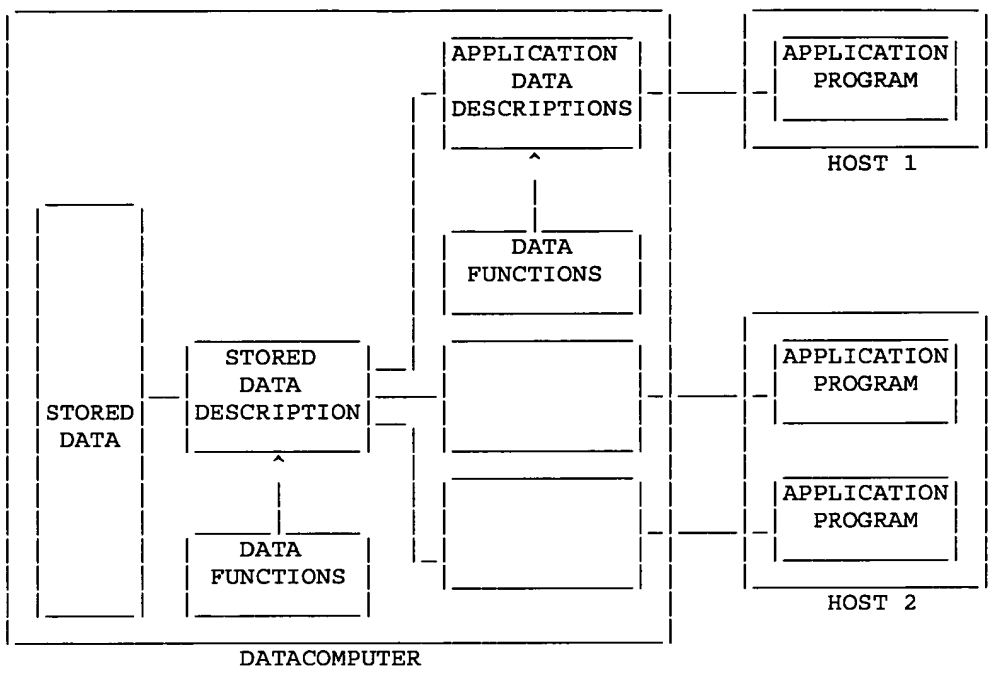


Figure 2-2
Multiple User Interaction with the Datacomputer

2.7.3 Communication Model

We intend that datalanguage, while at a high level conceptually, will be at a low level syntactically. Datalanguage provides a set of primitive functions, and a set of commonly used higher level functions (see section 4 on the datalanguage model). In addition, users can define their own functions so that they can communicate with the datacomputer at a level as conceptually close to the application as possible.

There are two reasons for datalanguage being at a low level syntactically. First, it is undesirable to have programs composing requests into an elaborate format only to be decomposed by the datacomputer. Second, by choosing a specific high level syntax, the datacomputer would be imposing a set of conventions and terminology which would not necessarily correspond to those of most users.

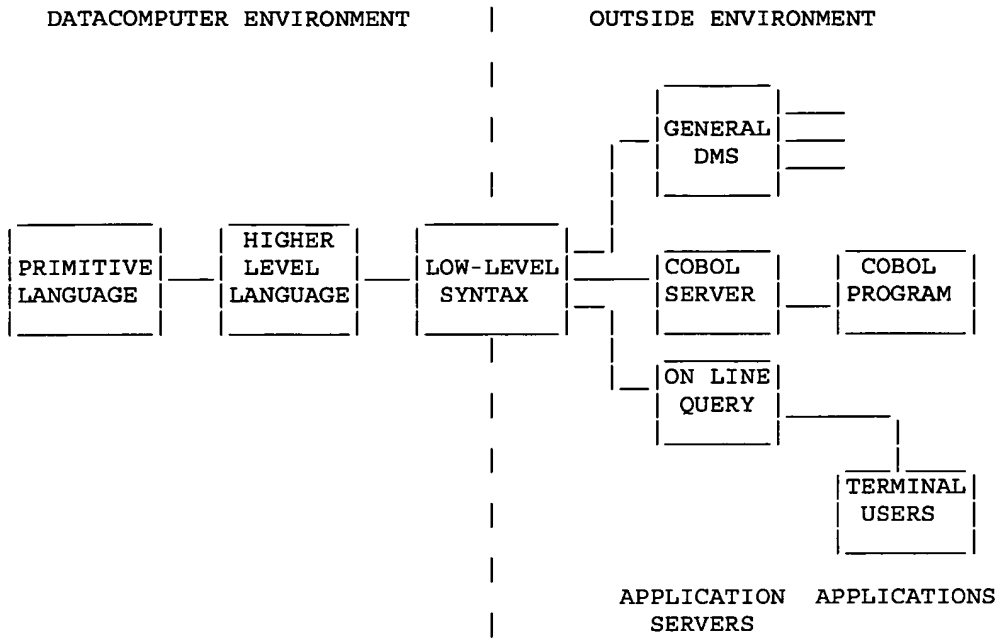


Figure 2-3
Datacomputer/User Working Environment

2.8 Summary

In this section we have presented the major considerations which have influenced the current datalanguage design effort. The datacomputer has much in common with most large-scale shared data management systems, but also has a number of overriding concerns unique to the datacomputer concept. The most important of these are the existence of a separate box containing both hardware and software, the control of an extremely large storage device, and embedding in a computer network environment. Data sharing in such an environment is a central concern of the design. Both extensive data description facilities and high level communication

between user and datacomputer are necessary for data integrity and for datacomputer optimization of user requests. In addition, the expected use of the datacomputer involves satisfying several conflicting constraints for different modes of operation. One way of satisfying various user needs is to provide datalanguage features so that users may develop their own application packages within datalanguage.

3. Principal Language Concepts

This section discusses the principal facilities of datalanguage. Specific details of the language are not presented, however, the discussion includes the motivation behind the inclusion of the various language features and also defines, in an informal way, the terms we use.

3.1 Basic Data Items

Basic data are the atomic level of all data constructions; they cannot be decomposed. All higher level data structures are fundamentally composed of basic data items. Many types of basic data items will be provided. The type of an item determines what operations can be performed on the item and the meaning of those operations. Datalanguage will provide those primitive types of data items which are commonly used in computing systems to model the real world.

The following basic types of data will be available in datalanguage: _fixed_point_numbers_, _floating_point_numbers_, _characters_, _booleans_, and _bits_. These types of items are "understood" by the datacomputer system to the extent that operations are based on the type of an item. Datalanguage will also include an _uninterpreted_ type of item, for data which will only be moved (including transmitted) from one place to another. This type of data will only be understood in the trivial sense that the datacomputer can determine if two items of the uninterpreted type are identical. Standard operations on the basic types of items will be available. Operations will be included so that the datacomputer user can describe a wide range of data management functions. They are not included with the intent of encouraging use of the datacomputer for the solving of highly computational problems.

3.2 Data Aggregates

Data aggregates are compositions of basic data items and possibly other data aggregates. The types of data aggregates which are provided allow for the construction of hierarchical relationships of data. The aggregates which will definitely be available are classified as _structs_, _arrays_, _strings_, _lists_, and _directories_.

A struct is a static aggregate of data items (called _components_). A struct is static in the sense that the components of a struct cannot be added or deleted from the struct, they are inextricably bound to the struct. Associated with each component of the struct is a name by which that component may be referenced relative to the struct. The struct aggregate may be used to model what is often thought of as a record,

with each component being a field of that record. A struct can also be used to group components of a record which are more strongly related, conceptually, than other components and may be operated on together.

Arrays allow for repetition in data structures. An array, like a struct, is a static aggregate of data items (called _members_). Each member of an array is of the same type. Associated with each member is an index by which that member can be referenced relative to the array. Arrays can he used to model repeating data in a record (repeating groups).

The concept of string is actually a hybrid of basic data and data aggregates. Strings are aggregates in that they are compositions (similar to arrays) of more primitive data (e.g., characters). They are, however, generally conceived of as basic in that they are mostly viewed as a unit rather than as a collection of items, where each item has individual importance. Also the meaning of a string is highly dependent on the order of the individual components. In more concrete terms, there are operations which are defined on specific types of strings. For example, the logical operators (_and_, _or_, etc.) are defined to operate on strings of bits. However, there are no operations which are defined on arrays of bits, although there are operations defined on both arrays, in general, and on bits. Strings of characters, bits, and uninterpreted data will be available in datalanguage.

Lists are like arrays in that they are collection of similar members. However, lists are dynamic rather than static. Members of a list can be added and deleted from the list. Although, the members of a list are ordered (in fact more than one ordering can be defined on a list), the list is not intended to be referenced via an index, as is the case with an array. Members of a list can be referenced via some method of sequencing through the list. A list member, or set (see discussion under virtual data) of members, can also be referenced, by some method of identification by content. The list structure can be used to model the common notion of a file. Also restrictive use of lists as components of structs provides power with respect to the construction of dynamic hierarchical data relationships below the file level. For example, the members of a list may themselves be, in part, composed of lists, as in a list of families, where each family contains a list of children as well as other information.

Directories are dynamic data aggregates which may contain any type of data item. Data items contained in a directory are called _nodes_. Associated with each node of a directory is a name by which that data item can be referenced relative to the directory. As with lists, items may be dynamically added to and deleted from a directory. The primary motivation behind providing the directory capability is to allow the user to group conceptually related data together. Since directories

need not contain only file type information, "auxiliary" data can be kept as part of the directory. For example, "constant" information, like salary range tables for a corporation data base; or user defined operations and data types (see below) can be maintained in a directory along with the data which may use this information. Also directories may themselves be part of a directory, allowing for a hierarchy of data grouping.

Directories will also be defined so that system controlled information can be maintained with some of the subordinate items (e.g. time of creation, time of update, privacy locks, etc.). It may also be possible to allow the data user to define and control his own information which would be maintained with the data. At the least, the design of datalanguage will allow for parametric control over the information managed by the system.

Directories are the most general and dynamic type of aggregate data. Both the name and description (see below) of directory nodes exist with the nodes themselves, rather than as part of the description of the directory. Also the level of nesting of a directory is dynamic since directories can be dynamically added to directories. Directories are the only aggregate for which this is true.

Datalanguage will also provide some specific and useful variations of the above data aggregates. Structs will be available which allow for optional components. In this case the existence of a component would be based on the contents of other components. It may also he possible to allow for the existence to be based on information found at a higher level of data hierarchy. Similarly, components with _unresolved_ type will be provided. That is the component may be one of a fixed number of types. The type of the component would be based on the contents of other components of the struct. It is also desirable to allow the type or existence of a component to be based on information other than the contents of other components. For instance, the type of one component might be based on the type of another component. In general, we would like for datalanguage to allow for the attributes (see below) of one item to be a function of the attributes of other items.

We would also like to provide mixed lists. Mixed lists are lists which contain more than one type of member. In this case the members would have to be self defining. That is, the type of all member would have to be "alike" to the degree that information which defines the type of that member could be found.

Similar to components whose type is unresolved are Arrays with unresolved length. In this case, information defining the length of the array must be carried with the array or perhaps with other components of an aggregate which encompasses the array.

In all of the above cases the type of an item is unresolved to some degree and information which totally resolves the type is carried with the item. It is possible that in some or perhaps all of these cases the datacomputer system could be responsible for the maintenance of this information, making it invisible to the data user.

3.3 General Relational Capabilities

The data aggregates described above allow for the modeling of various relationships among data. All relationships which can be constructed are hierarchical.

Two approaches can he taken to provide the capability of modeling non-hierarchical relationships. New types of data aggregates can be introduced which will broaden the range of data relationships expressible in datalanguage. Or, a basic data type of "pointer" can be introduced which will serve as a primitive out of which relations can be represented. Pointer would be a data type which establishes some kind of correspondence from one item to another. That is, it would be a method of finding one item, given another . Providing the ability to have items of type pointer does not necessitate the introduction of the concept of address which we deem to be a dangerous step. For example, an item defined to point to a record in a personnel file could contain a social security number which is contained in each record of the file and uniquely identifies that record. In general a pointer is an item of information which can be used to uniquely identify another item.

While the pointer approach provides the greater degree of flexibility, it does this at the price of relegating much of the work to the user as well as severely limiting the amount of control the datacomputer system has over the data. A hybrid solution is possible, where some new aggregate data types are provided as well as a restricted form of pointer data type. While the approach to be taken is still being studied, the datalanguage design will include some method of expressing non-hierarchical data structures.

3.4 Ordering of Data

Lists are generally viewed as ordered. It is possible, however, that a list can be used to model a dynamic collection of similar items which are not seen as ordered. The unordered case is important, in that, given this information the datacomputer can be more efficient since new members can be added wherever it is convenient.

There are a number of ways a list can be ordered. For instance, the ordering of a list can be based on the contents of its members. In the

simplest case this involves the contents of a basic data item. For example, a list of structs containing information on employees of a company may be ordered on the component which contains the employee's social security number. More complex ordering criteria are possible. For example, the same list could be ordered alphabetically with respect to the employee's last name. In this case the ordering relation is a function of two items, the last and first names. The user might also want to define his own ordering scheme, even for orderings based on basic data items. An ordering could be based on an employee's job title which might even utilize auxiliary data (i.e. data external to the list). It is also possible to maintain a list in order of insertion. In the most general case, the user could dynamically define his ordering by specification of where an item is to be placed as part of his insertion requests. In all of the above cases, data could be maintained in ascending or descending order.

In addition to maintenance of a list in some order, it is possible to define one or more orderings "imposed" on a list. These orderings must be based on the contents of a list's members. This situation is similar to the concept of virtual data (see below) in that the list is not physically maintained in a given order, but retrieved as if it were. Orderings of this type can be dynamically formed (see discussion of set under virtual data). Imposed orderings can be accomplished via the maintenance of auxiliary structures (see discussion under internal representation) or by utilization of a sorting strategy on retrievals. Much work has been done with regard to effective implementation of the maintenance and imposition of orderings on lists. This work is described in working paper number 2.

3.5 Data Integrity

An important feature of any data management system is the ability to have the system insure the integrity of the data. Data needs to be protected against erroneous manipulation by people and against system failure.

Datalanguage will provide automatic validity checks. Many flavors need to be provided so that appropriate trade-offs can be made between the degree of insurance and the cost of validation. The datalanguage user will be able to request constant validation: where validity checks are made whenever the data is updated; validation on access: where validity checks are performed when data is referenced but before it is retrieved; regularly scheduled validation: where the data is checked at regular intervals; background validation: where the system will run checks in its spare time; and validation on demand. Constant validation and validation on access are actually special cases of the more general concept of event triggered validation. In this case the user specifies

an event which will cause data validation procedures to be invoked. This feature can be used to accomplish such things as validation following a "batch" of updates. Also, some mechanism for specifying combinations of these types would be useful.

In order for some of the data validation techniques to be effective, it may be necessary to keep some data validation "bookkeeping" information with the data. For example, information which can be used to determine whether an item has been checked since it was last updated might be used to cause validation on access if there has not been a recent background validation. The datacomputer may provide for optional automatic maintenance of such special kinds of information.

In order for the datacomputer system to insure data validity, the user must define what valid is. Two types of validation can be requested. In the first case the user can tell the datacomputer that a specific data item may only assume one of a specific set of values. For example, the color component of a struct may only assume the values 'red', 'green', or 'blue'. The other case is where some relation must hold between members of an aggregate. For example, if the sex component of a struct is 'male' then the number of pregnancies component must be 0.

Data validation is only half of the data integrity picture. Data integrity involves methods of restoring damaged data. This requires maintenance of redundant information. Features will be provided which will make the datacomputer system responsible for the maintenance of redundant data and possibly even automatic restoration of damaged data. In section 2 we discussed possible uses of the datacomputer for file backup. All features which are provided for this purpose will also be available as methods of maintaining backup information for restoration of files residing at the datacomputer.

3.6 Privacy

Datalanguage will have to provide extensive privacy and protection capabilities. In its simplest form a privacy lock is provided at the file level. The lock is opened with a password key. Associated with this key is a set of privileges (reading, updating, etc.). Two degrees of generality are sought. Privacy should be available at all levels of data. Therefore, groups of related data, including groups of files could be made private by creating private directories. Also, specific fields of records could be made private by having private components of a struct where other components of the struct are visible to a wider (or different) class of users. We would also like the user to be able to define his own mechanism. In this way, very personalized, complex, and hence secure mechanisms can be defined. Also features such as 'everyone can see his own salary' might be possible.

3.7 Conversion

Many types of data are related in that some or all of the possible values of one type of data have an "obvious" translation to the values of another. For example, the character '6' has a natural translation to the integer 6, or the six character string 'abc ' (three trailing blanks) has a natural translation to the four character string 'abc ' (one trailing blank). Datalanguage will provide conversion capabilities for the standard, commonly called for, translations. These conversions can be explicitly invoked by the user or implicitly invoked when data of one type is needed for an operation but data of another type is provided. In the case of implicit invocation of conversion of data the user will have control over whether conversion takes place for a given data item. More generally we would like to provide a facility whereby the user could specify conditions which determine when an item is to be converted. Also, the user should be able to define his own conversion operations, either for a conversion between types which is not provided by the datacomputer system or to override the standard conversion operation for some or all items of a given type.

3.8 Virtual and Derived Data

Often, information important to users of data is embedded in that data rather than explicitly maintained. For example, the dollar value of an individual's interest in a company in a file of stock holders. Since the value of the company changes frequently, it is not feasible to maintain this information with each record. It is useful to be able to use the file as if information of this type was part of each record. When referencing the dollar value field of a record, the datacomputer system would automatically use information in the record, such as percentage of ownership in the company, possibly in conjunction with information which is not part of the record but is maintained elsewhere, such as company assets, to compute the dollar value. In this way the data user need not be concerned with the fact that this information is not actually maintained in the record.

The _set_, which is a specific type of virtual container in datalanguage, deserves special mention. A set is a virtual list. For example, suppose there is a real list of people representing some population sample. By real (or actual) data we mean data which is physically stored at the datacomputer. A set could be defined to contain all members of this list who are automobile owners. The set concept provides a powerful feature for viewing data as belonging to more than one collection without physical duplication. Sets are also useful, in that, they can be dynamically formed. Given an actual list, sets based on that list can be created without having been previously described.

As mentioned above, virtual data can be very economical. These economies may become most important with respect to the use of sets. Savings are found not only in regard to storage requirements, but also in regard to processing efficiency. Processing time can be reduced as a result of calculations being performed only when the data is accessed. The ability to obtain efficient operation by optimization becomes greater when virtual data is defined in terms of other virtual data. For sets, large savings may be realized by straight forward "optimization" of the nested calculations.

The above ideas are made more clear by example. Having created a set of automobile owners, A, a set of home owners, HA, can be defined based on A. The members of HA can be produced very efficiently, in one step, by retrieving people who are both automobile owners and home owners. This is more efficient than actually producing the set, A and then using it to create HA. This is true when one or both pieces of information (automobile ownership and home ownership) are indexed (see discussion under internal representation) as well as when neither is indexed.

The same gains are achieved when operations on virtual data are requested. For example, if a set, H, had been defined as the set of homeowners based on the original list of people, the set, HA, could have been defined as the intersection (see discussion on operators) of A and H. In this case too, HA can be calculated in one step. Use of sets allows the user to request data manipulations in a form close to his conceptual view, leaving the problem of effective processing of his request to the datacomputer.

Another use of virtual data is to accomplish data sharing. An item could be defined, virtually, as the contents of another item. If no restriction is placed on what this item can be, we have the ability to define two paths of access to the same data. Hence, data can be made subordinate to two or more aggregate structures. Stated another way, there are two or more paths of access to the data. This capability can be used to model data which is part of more than one data relationship. For example, two files could have the same records without maintaining duplicate copies.

It will also be possible, via data sharing to look at data in different ways. Shared data might behave differently depending on how (and ultimately by whom) it is accessed. Although, the ability to have multiple paths to the same data and the ability to have data which is calculated on access are both part of the general virtual data capability, datalanguage will probably provide these as separate features, since they have different usage characteristics.

Derived data is similar to virtual data in that it is redundant data which can be calculated from other information. Unlike virtual data it

is physically maintained. The user can choose between virtual and derived data as a result of considering trade-offs based on: estimated cost of calculation; frequency of update; estimated cost of storage; and frequency of access. For example, suppose a file contains a list of budgets for various projects in a department. The departmental budget can be calculated as a function of the individual project budgets. This information might be defined as derived data since it is expected to be updated infrequently (e.g., once a year), while it is expected to be accessed relatively often.

Options will be provided which give the user control with regard to when the calculation of derived data is to be done. These options will be similar to those provided for control of data validity operations. The data validation and derived data concepts are similar in that some operation must be performed on related data. In the case of data validation, the information derived is the condition of data.

3.9 Internal Representation

To this point, we have discussed only the high level, logical, aspects of data. Since data, at any given time, must reside on some physical device a representation of the data must be chosen. In some cases it is appropriate to leave this choice to the datacomputer system. For example, the representation of information which is used in the process of transmitting other data, but which itself resides solely at the datacomputer may not be of any concern to the user.

However, it is important that the user be capable of controlling the choice of representation. In any application which requires mostly transmission of data rather than interpretation of the data by the datacomputer, the data should be maintained in a form consistent with the system which communicates with the datacomputer. With respect to basic types of data, datalanguage will provide most representations commonly used in systems with which it interacts. For some types (e.g., fixed point) this will be accomplished by providing for parametric (e.g., sign convention, size) description of the representation. In other cases (e.g., floating point) specific representations will be offered (e.g., system 360 short floating point, system 360 long floating point, pdp-10 floating point, etc.).

Another aspect of the internal representation problem regards aggregate structures. The method chosen to represent aggregate structures may largely affect the cost of manipulating the data. The user must have control over this representation since only he has any idea of how the data is to be used. Datalanguage will provide a variety of representational options which will allow for efficient implementation of data structures. This includes the availability of auxiliary

structures, automatically maintained by the data computer system. These structures can be used to effect efficient retrieval of subsets of data collections based on the contents of the members (i.e. the common concept of indices), efficient maintenance of orderings on a collection of data, maintenance of redundant information for the purpose of data integrity, and efficient handling of shared data whose behavioral characteristics are dependent on the path of access. It should be noted here that, the datalanguage design effort, will attempt to provide methods whereby the data user can describe the expected use of his data, so that details of internal representation can be left to the datacomputer.

3.10 Data Attributes and Data Classes

The type of an item determines the operations which are valid on that item and what they mean. _Data_attributes_ are refinements on the type of data. The data attributes affect the meaning of operations. For example, we would like to provide for the option of defining fixed point items to be scaled. The scale factor, in this case, would be an attribute of fixed point data. It effects the meaning of operations on that data. The attribute concept is useful in that it allows information concerning the manipulation of an item to be associated with the item rather than with the invocation of all operations on that item.

The attribute concept can be applied to aggregate as well as basic data. For example, one attribute of a list could define where a new member is to be inserted. Options might be: insert at the beginning of the list; insert at the end of the list; or insert in some order based on the contents of the member. Adding a new member to a list with one of the above attributes could be done by issuing a simple insert request without having to specify where the new member is to be inserted.

The _data_class_ concept is actually the inverse of the data attribute concept. A data class is a collection of data types. The data class concept allows for definition of operations, independent of specific type of an item. For example, by defining the data class arithmetic to be composed of fixed point and floating point types of data, the comparison operators (_equal_, _less_than_, etc.) can be defined to operate on arithmetic data, independent of whether it is fixed or floating point. Also the concept of data aggregate can be seen as a class encompassing directories, lists, etc. As there are operations defined on arithmetic data, there are also operations defined on arbitrary aggregates.

The inverse relationship between data classes and data attributes is very strong. For example, the concept of list can be seen as a data class, encompassing all types of lists (e.g., lists of integers, lists

of character strings, etc.), independent of the types of their members. The type of a list's members (e.g., integer, character string, etc.) are then seen as attributes. Data attributes and classes are also relative concepts. While the concept of list can be viewed as a data class, it can also be seen as an attribute, relative to the concept of data aggregate.

3.11 Data Description

A _data_description_ is a statement of the properties (see discussion of attributes) of a data item. Examples of properties which are recorded in a description are: the name of an item; its size; its data type; its internal representation; privacy information; etc.

Datalanguage will contain mechanisms for specifying data descriptions. These descriptions will be processed by the data computer, and used whenever the data item is referenced. The user will be able to physically create data only by first specifying their descriptions. The properties of a description can be divided into groups according to their function. Some have the function of specifying details of representation, which will not be of interest to most users, while others, such as the name are of almost universal interest.

All user data is a part of some larger (user or system) data structure. The structures containing data establish a path of access to the data. In the process of following this path the datacomputer system must accrue a complete description of the data item. For example, the description of a data item of a directory may be found associated with that node of the directory. Members of a list or array are described as part of the description of the list or array. We must dispose of two seeming exceptions. First, while aspects of data may (on user request) be left to the system, those aspects are still described, they are described by the system. As discussed above, some data will be, to some degree, self describing (e.g. members of mixed lists). However, it is fully described in some encompassing structure, in that a method of determining the full description is described.

It is worth noting here that the sooner a complete description is found in the path of access, the more effective the datacomputer is likely to be in processing requests which manipulate a data item. However, the ability to have data whose complete description does not exist at high levels of the access path provides greater flexibility in the definition of data structures.

3.12 Data Reference

Data cannot be manipulated unless it can be referenced. In the same way that data cannot exist without its being described, it cannot exist unless there is a path of access to the data. The method of data reference is to define the path of access to the data. As mentioned above, there is a method of referencing any item relative to the data aggregate which contains it. Nodes of directories and components of structs are referenced via the name associated with the node or component. Members of arrays are referenced via the index associated with the member. Members of lists are referenced via some method of specifying the position of the member or by uniquely identifying the member by content. To reference any arbitrary data item the path of access must be fully defined by either explicit or implicit definition of each link in the chain. In the case of virtual data there is an extra implicit link in the chain, that being the method employed to obtain the data from other data items. It should be noted also that if pointers are provided (see discussion on general relational capabilities) they can also serve as a link in the chain of access to an item.

The design of datalanguage will ease the problem (and reduce the cost) of referencing data items by providing methods whereby part of the access path can be implicitly defined. For example, datalanguage will provide a concept of "context". During the course of interacting with the datacomputer, levels of context can be set up so that data can be referenced directly, in context. For example, on initiating a session the user may (in fact will probably be required to) define a directory which will be the context of that session. All items subordinate to this directory can be referenced directly in this context. Another feature will be partial qualification. Each level of struct need not be mentioned in order to reference an item embedded in a deep nest of structs. Only those intermediate levels which are sufficient to uniquely identify the item need be specified.

3.13 Operations

In this section we discuss the builtin functions of datalanguage which are of central importance in manipulating data. Functions which operate on items, functions which operate on aggregates, primitive functions and high-level functions are discussed.

Of the primitives which operate on items, those of most interest are assignment, comparisons, logicals, arithmetics and conversion functions.

Primitive assignment transfers a value from one item to another; these items must be of the same type. When they are of different types,

either conversion must be performed, or some non-primitive form of assignment is involved.

The comparison operators accept a pair of items of the same type, and return a boolean object which indicates whether or not a given condition obtains. The type determines how many different conditions can be compared for. A pair of numeric items can be compared to see which is greater, while a pair of uninterpreted items can be compared only for equality. In general, a concept of "greater than" is builtin for a datatype only if it is a very widely applied concept. The comparison operators are used in the construction of inclusion conditions when defining subsets of aggregate data.

The result of a comparison operation is a boolean item: one whose value is either TRUE or FALSE. Logical primitives are provided and generalized boolean functions can be constructed from them. With logical and comparison operators, complex conditions for inclusion of objects in sets can be specified.

Arithmetic operators will be available for the manipulation of numeric data. Here, we are not interested in generalized computation, but in applications of arithmetic in data selection, space allocation, subscript calculation, iteration control, etc.

Conversion is an important part of generalized data translation, and we are interested in providing a substantial builtin conversion facility. In particular, we will want to provide an efficient system routine for each "standard" or widely-used conversion function. Of particular importance are conversions to and from character string data; in character string representation of, for example, numeric items, there are many possible formats corresponding to a single data type. Conversion between character sets and dealing with padding and truncation are viewed as conversion problems.

There are two principal classes of primitive operators defined on aggregates: those related to data reference (see previous section) and those which add and delete components. Changing an existing component is accomplished through assignment, and is an operation on the component, not the aggregate.

Addition and deletion of components is defined only for aggregates which are not inherently static in composition. Thus one can add a component to a LIST, but not to an ARRAY. To specify deletion it is necessary to specify which component is to be deleted, and from which aggregate (in the case that it is shared). Addition requires specification of new component, aggregate, and sometimes auxiliary information. For example, some aggregate types would permit addition of new components anywhere in the structure; in these a position must be indicated, relative to any

existing components.

Often it is desirable to operate on some of the members of a list, or to treat a group of members as a list in its own right. For example, it might be common to transmit to a remote program for analysis, the medical history of patients developing heart disease before the age of 30. These may be just a few of the members of a large list of patients.

In this case, the operation to be performed is transmission to the remote system; this operation is performed on several members of the list of patients. The ones to be transmitted are thought of as a _set_; the set is specified as containing all the members of a given list satisfying two conditions: (1) age less than 30, and (2) has heart disease.

Sets can be defined explicitly, or implicitly simply with appropriate reference mechanisms. _Definition_ of a set is distinct from _identification_of_membership_, which is distinct from _access_to_membership_. Definition involves specifying the candidates for set membership and specifying a rule by which members of the set can be distinguished from non-members; for example, an inclusion condition such as "under 30 with heart disease". Identification involves effective application of the rule to all candidates for membership. When the membership has been identified, it can be counted, but the data itself has not necessarily been accessed. When a member is accessed, its contents can be operated on.

Primitives to accomplish each of these operations on a set will be provided; however, it will ordinarily be optimal for the datacomputer to determine when each step should be performed. To enable users to operate at a level at which the datacomputer can optimize effectively, higher-level operators on sets will be provided. Some of these are logical operators, such as union and intersection. These input and output sets. Also available is an operator which complements a set (since the definition establishes all possible candidates, a set always has a well-defined complement).

These higher level operators can be applied to any defined set; the set members need not be identified or accessed. The system will perform such operations without actually accessing members if it can.

Some of the other operators on sets are counting membership, partitioning a set into a set of sets, uniting a set of sets into a set. A set can be used to reference another set, providing there is a well-defined way to identify members of the second set given the first set. For example, a set C may contain all the children doing poorly in school. A set F may be defined, where the members of F are the records about families having a child in set C.

Some other useful operations on sets are: adding all the members of a set to an aggregate, deleting all the members of a set (frequently such a massive change can be performed far more efficiently than the same set of changes individually requested), changing all the members of a set in a given way.

A set can be made into a list, by actually accessing each member and physically collecting them.

Some of the operations on lists are: concatenation of lists into larger lists, division of a list into smaller lists, sorting a list, merging a pair of ordered lists (preserving order).

This is not intended to be a full enumeration of high-level operations, but to be suggestive. We are planning to build in high-level functions for operations which are used very commonly, and can be implemented within the system significantly better than they can be implemented by users in the language. For most of the functions mentioned here, considerable knowledge is accumulated on good implementations. In particular, the techniques used for inverted file access provide many set operations to be performed without actual access to the data.

3.14 Control

The control features of datalanguage are to the basic operations as data aggregates are to the basic data items. Control features are used to create complex requests out of the basic requests provided by datalanguage.

Conditional requests allow the user to alter the normal request flow by specifying that certain requests are to be executed under certain conditions. In general datalanguage will provide the ability to chose at most one of a number of requests to be made based on some set of conditions or the value of some item. In its simplest form the conditional allows for optional execution of a given request.

Iterative requests cause a request (called the body) to be executed a fixed or variable number of times or until a given condition is met. Datalanguage will provide iterative requests that will allow for similar manipulations to be performed on all members of some aggregate structure as well as the standard type of iterative request based on counters. By providing a capability of directly expressing manipulations on aggregates which require processing all of the items subordinate to the aggregate, the datacomputer can be more efficient in processing user requests. For example, a user defined conversion process which operates on character strings, can be implemented far more efficiently if the datacomputer is explicitly informed that the process requires sequential

processing of the characters. Datalanguage will also provide for parallel iteration. For example, the user will be able to specify operations which require sequencing through two or more lists in parallel. This would be done if the contents of one file were to be updated based on a file of correction information.

Compound requests are collections of requests which act as one. They are primarily provided to allow for the conditional performance of or iteration on more than one statement. Compound requests also provide request reference points which can be used to control the request processing flow. That is, compound requests can be "named". The datalanguage user will be able to specify control information which will conditionally cause a compound request to be exited. By providing naming, the user may cause any number of previously entered compound requests to be exited.

We do not intend to provide the traditional _goto_ capability. By not including a goto request, the chances for efficient operation (via optimization) of the datacomputer are increased. We also hope, in this way, to force the datalanguage user to specify his data manipulations in a clear style.

Two forms of the compound request will be provided, ordered and unordered. In the unordered case the user is informing the datacomputer that the requests can be performed in any order. This should allow the datacomputer to perform more efficiently and might even allow for parallel processing.

During a session with the datacomputer it is likely that a user will find a need for temporary data. That is, data which is used to remember, for a short term, information which is needed for the processing of requests. This short term might be a session or a small part of a session. Datalanguage will provide a temporary data facility. Temporary data will be easy to create, use and dispose of. This will be accomplished by allowing the system to (optionally) make many decisions regarding the data. For example the representation of a temporary integer item will often be of no concern to the user. Some features which are provided for permanent data will be deemed irrelevant with regard to temporary data.

Temporary data will be associated with a collection of requests in what will be called a block. A block will be no different than a compound request with the exception that data is defined with the requests which compose it and is automatically created on entrance to the block and destroyed on exiting the block.

3.15 Extensibility

The goals of datalanguage are to provide facilities of data structure at two levels. At one level the user may take advantage of high level data capabilities which will do much of his data management work automatically and which allows for the data computer to operate more effectively in some cases since it has been given control of the data. At another level, however, features are provided which allow the user to describe his application in terms of primitive concepts. In this way the datacomputer user may compose a large variety of data constructs and has great flexibility with respect to the manipulations he can perform on his data. Also by interacting with the datacomputer at the primitive level, the user can exercise a good deal of control over the methods employed by the datacomputer which may result in more effective usage of resources for non-standard applications. Datalanguage will provide features which allow the user to create an environment whereby the datacomputer system appears to provide features especially tailored to his application.

The control features discussed above allow the user to extend the operations available on data by appropriate composition of the operations. Datalanguage will provide a method of defining a composite request to be a new request (called a _function_). In this way a new operation on specific data can be defined once and then used repeatedly. In order that the user may define general operations, datalanguage will provide functions which can be parameterized. That is, functions will not only be able to operate on specific data but may be defined to work on any data of a specific type. This capability will not be limited to basic data types (e.g. integers) or even specific aggregate types (e.g. array of integers) but will also include the ability to define functions which operate on classes of data. For example, functions can be defined which operate on lists independent of the type of the list members. Also provided, will be the ability to expand and modify existing functions as well as creating new functions. This includes expanding the types of data for which a function is defined or modifying the behavior of a function for certain types of data.

As with operations, the data aggregates discussed above allow the user to extend the primitive data types by appropriate composition. For example, a two dimensional array of integers can be created by creating an array of arrays of integers. The situation for data types is analogous to that of operations. Datalanguage will provide the ability to define a composition of data to be a new data type. Also the capability of defining general data structures will be provided by essentially parameterizing the new data definition. This would allow the general concept of two dimensional array to be defined as an array of arrays. Once defined, one could create two dimensional arrays of integers, two dimensional arrays of booleans, etc. As with functions

there is also a need to expand or modify existing data types. One might want to expand the attributes which apply to a given data type, in that he might want to add new attributes, or add new choices for the existing attributes.

The control features can be extended also. Special control features might be needed to process a data structure in a special way or to process a user defined data structure. For example, if a tree type data structure has been defined in terms of lists of lists, the user might like to define a control function which causes a specified operation to be performed on each item of a specified tree. As with data types and functions, there is a need to be able to modify and extend existing control features as well as the ability to create new ones.

Datalanguage will provide the ability to treat data descriptions and operations in much the same way that data is treated. One can describe and manipulate descriptions and operations in the same way that he can describe and manipulate data. It is impossible to talk about data types without consideration of operations and equally as impossible to talk about operations without an understanding of the data types they operate on. In order for the user to be able to effect the behavior of the datacomputer system, the design of datalanguage will include a definition of the operational cycle of the datacomputer. Precise definitions of all aspects of data (data attributes, data classes, relationship of aggregates to their subordinate items, etc.) in terms of their interaction with datalanguage operations will be made. In this way the datacomputer can offer tools which will give the datacomputer user the ability to be an active participant in the design of the datalanguage which he uses.

4. A Model for Datalanguage Semantics

For the purpose of defining and experimenting with language semantics and with language processing techniques, we are developing a model datacomputer.

The principal elements of the model are the following:
(1) A set of primitive functions
(2) An environment in which data objects can be created, manipulated and deleted, using the primitives
(3) A structure for the representation of collections of data values, their descriptions, their relationships, and their names.
(4) An interpreter which executes the primitives
(5) A compiler which inputs requests in a very simple language, performs binding and macro expansion operations, and generates calls to the internal semantic primitives.

If our modeling efforts are successful, the model will evolve until it accepts a language like the datalanguage whose properties we have described in sections 2 and 3 of this paper. Then the process of writing the final specification will simply require reconciliation of details not modeled with structure that has been modeled. One rather large detail which we may never handle within the model is syntax; in this case reconciliation will be more involved; however, we firmly believe that the semantic structure should determine the syntax rather than the opposite, so we will be in the proper position to handle the problem.

By constructing a model for each of the elements listed above, we are "implementing" the language as we design it, in a very loose sense. In effect, we work in a laboratory, rather than working strictly on paper. Since we aren't concerned with the performance or usability of the datacomputer we are building in the laboratory, we are able to build without becoming involved with some of the most time-consuming concerns of an implementor. However, because we are building and tinkering, rather than simply working on paper, we do get some of the advantages that normally come with the experience of implementing one's ideas.

The model datacomputer is a program, developed in ECL, using the EL1 language. Presently we are interested in the process of developing the program, not running it. Our primary requirement is to have, in advance of the existence of datalanguage, a well-defined and flexible notation in which to specify data structures, function definitions and examples. EL1 is convenient for this. Having a program which actually works and acts like a simple datacomputer is really a by-product of specifying semantics in a programming language. It is not necessary for the program to work, but it does provide some nice features. It enhances the "laboratory" effect, by doing such things as automatically compiling

strings of primitives, displaying the state of the environment in complicated examples, automatically discovering inconsistencies (in the form of bugs), and so on.

There are two major reasons that EL1 is a convenient notation for specifying datalanguage semantics. One is that the languages have a certain amount in common, in both concepts and in goals in data description. (In part, this is because EL1 itself has been a good source of ideas in attacking the datalanguage problem). Both languages emphasize operations on data, independent of underlying representation. A second reason that EL1 is a convenient way to specify datalanguage, is that EL1 is extensible; in fact, many primitive functions could be embedded directly into EL1 by using the extension facilities. At times, we have chosen to embed less than we could, to expose problems of interest to us.

So far, the model has been useful primarily in exposing design issues and relationships between design decisions. Also, because it includes so many of the elements of the full system (compiler, interpreter, environment, etc.), it encourages a fairly complete analysis of any proposal.

In presenting the model in this section, we have chosen to emphasize ideas and examples, rather than formal definitions in EL1. This is because the ideas are more permanent and relevant at this point (the formalisms are changing rather frequently) and because we imagine people reading the formal definitions only to get at the ideas. The formal definitions may be interesting in themselves when the language is complete; at this point they are probably of interest only to us.

The section is organized into a large number of sub-sections. The first few are concerned with the basic concepts of data objects, descriptions, and relationships between objects. We then discuss primitive semantic functions and present informal definitions and examples in sections 4.7 and 4.8. Section 4.9 is a brief discussion of compilation, interpretation and the execution cycle. Section 4.10 provides a fairly elaborate example of how primitive functions can be combined to do something of interest: a selective retrieval by content. The last two sections wrap up with discussions of high-level functions and some conclusions.

4.1 Objects

An _object_ has a name, a description, and a value. It can be related to other objects.

The _name_ is a symbol, which can be used to access the object from

language functions.

The _description_ is a specification of properties of the object, many of which relate to the meaning or the representation of the value.

The _value_ is the information of ultimate interest in the object.

The relationships between objects are hierarchical. Each object can be related directly to at most four other objects, designated as its _parent_, its _child_, its _left_sibling_, and its _right_sibling_.

This specific concept of relationship is all that has been built in to the model to date. One of our primary objectives in the future is to experiment with more general relationships among objects.

4.2 Descriptions

A description has the components _name_, _type_ and _type-dependent_parameters_. It can be related hierarchically to other descriptions, according to a scheme similar to the one described for objects in 4.1.

The _name_ has a role in referencing, as in the case of objects.

Type is an undefined, intuitive idea for which we expect to develop a precise meaning within datalanguage(see section 3.10 for some of the ideas about this). In terms of the present model, it simply means one of the following: LIST, STRUCT, STRING, BOOL, DESC, DIR, FUNC, OPD. Each of these refers to a sort of value corresponding to common ideas in programming (with the exception of OPD, which is explained in section 4.7), and on which certain operations are defined.

Examples of _type-dependent_parameters are the two items needed to define a STRING: size option and size. A STRING is a sequence of characters; the size of the STRING is the number of characters in it. If a STRING has a fixed size, then size option is FIXED and size is the number of characters it always contains. If a STRING has a varying size, then size option is VARYING, and size is its maximum (clearly, it might also have a minimum in a more refined scheme).

When the description of an object has a type of STRING, it is commonly said that the object is a STRING.

4.3 Values

The value is the data itself.

An object of type BOOL can have only either the value TRUE or the value FALSE.

An object of type STRING has values such as 'ABC', 'JOHN', or 'BOSTON'.

Each value has a representation, in bits. Thus a BOOL is represented by a single bit, which will be a 'one' to represent TRUE and a 'zero' to represent FALSE.

4.4 Some examples

Here are some examples of structures involving objects, descriptions, and values. In these explanations and drawings, the objective is to convey some ideas about these primitive structures; considerable detail is omitted in the drawings in the interest of clarity.

Figure 4-1 shows two objects. X is of type string and has value 'ABC'. Y is of type bool and has value TRUE.

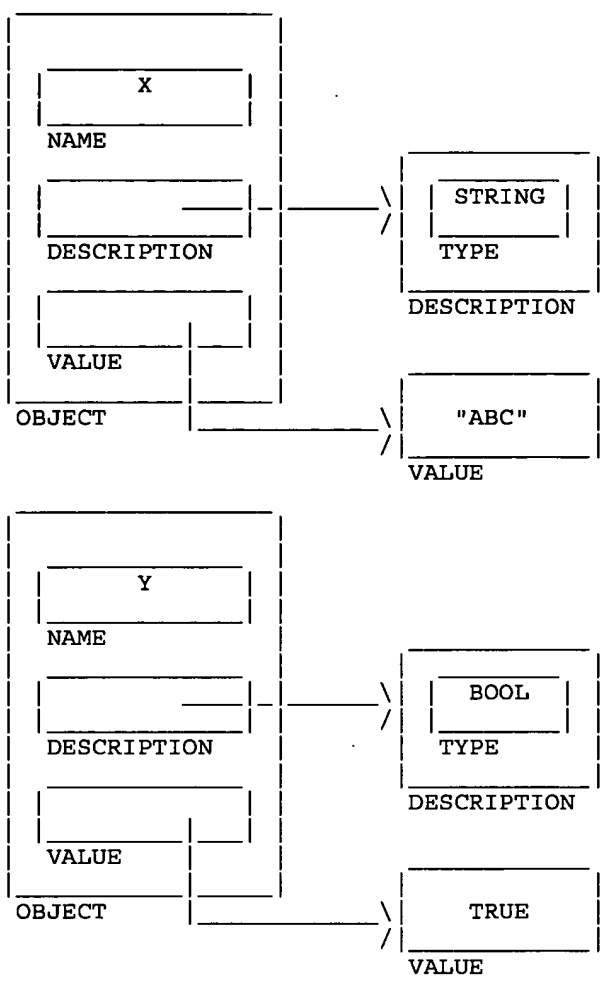


Figure 4-1
Two elementary objects

Figure 4-2 illustrates an object of type dir (a _directory_) and related objects. The directory has name SMITH. There are two objects entered in this directory, named X and Y.

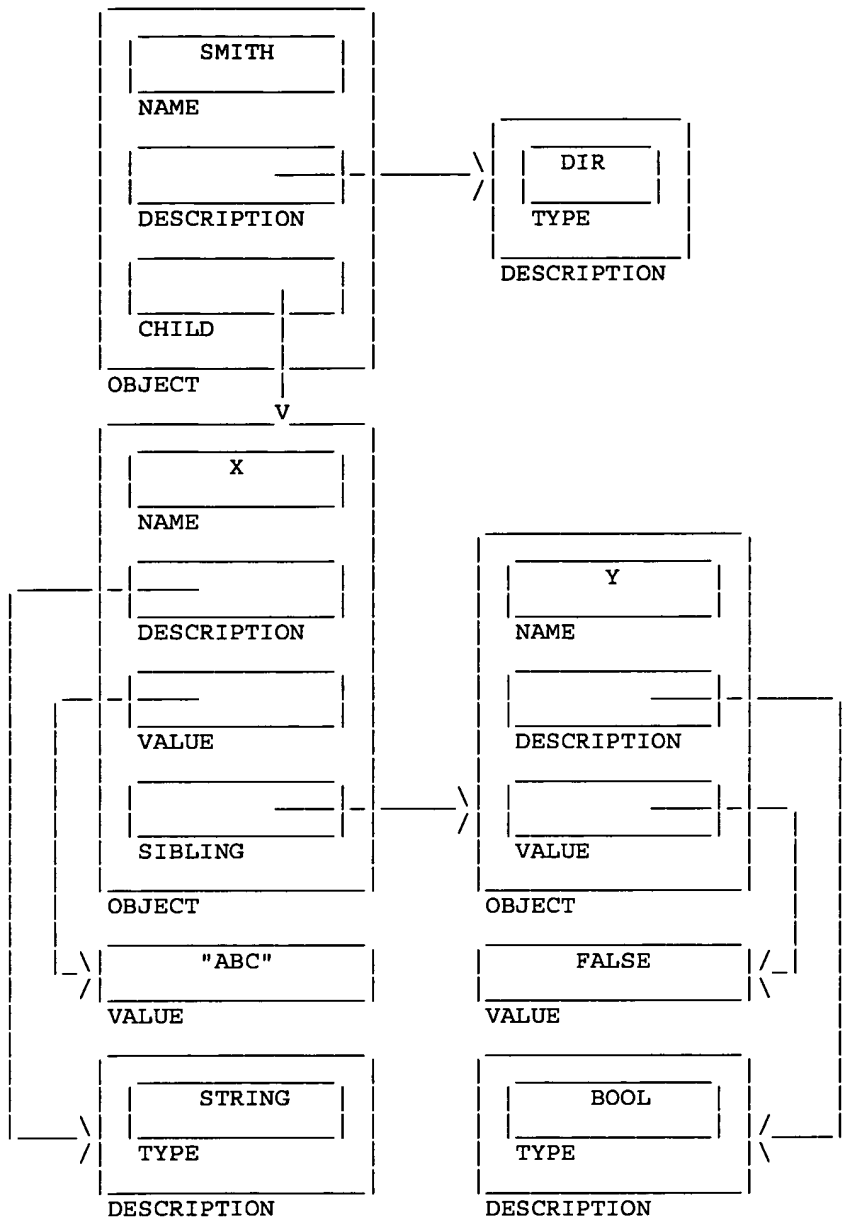


Figure 4-2: A directory with two members

The idea of a dir is similar to the idea of a file directory in most systems. A directory is a place where one can store named objects, freely adding and deleting them. The entries in the directory are all objects whose parent is that directory. Figure 4-3 shows a more rigidly structured group of objects. Here we have R, a struct, and A and B, a pair of strings. Note that the boxes labeled 'object' in figure 4-3 bear precisely the same relationships to one another as those labeled 'object' in 4-2. However, there are two conditions which hold for 4-3 but do not hold for 4-2: (1) the value of R contains the values of A and B, and (2) the descriptions of R, A and B are all related.

Structs have the following properties: (1) name and description of each component in the struct is established when the struct is created, and (2) in a value of the struct, the order of occurrence of component values is fixed.

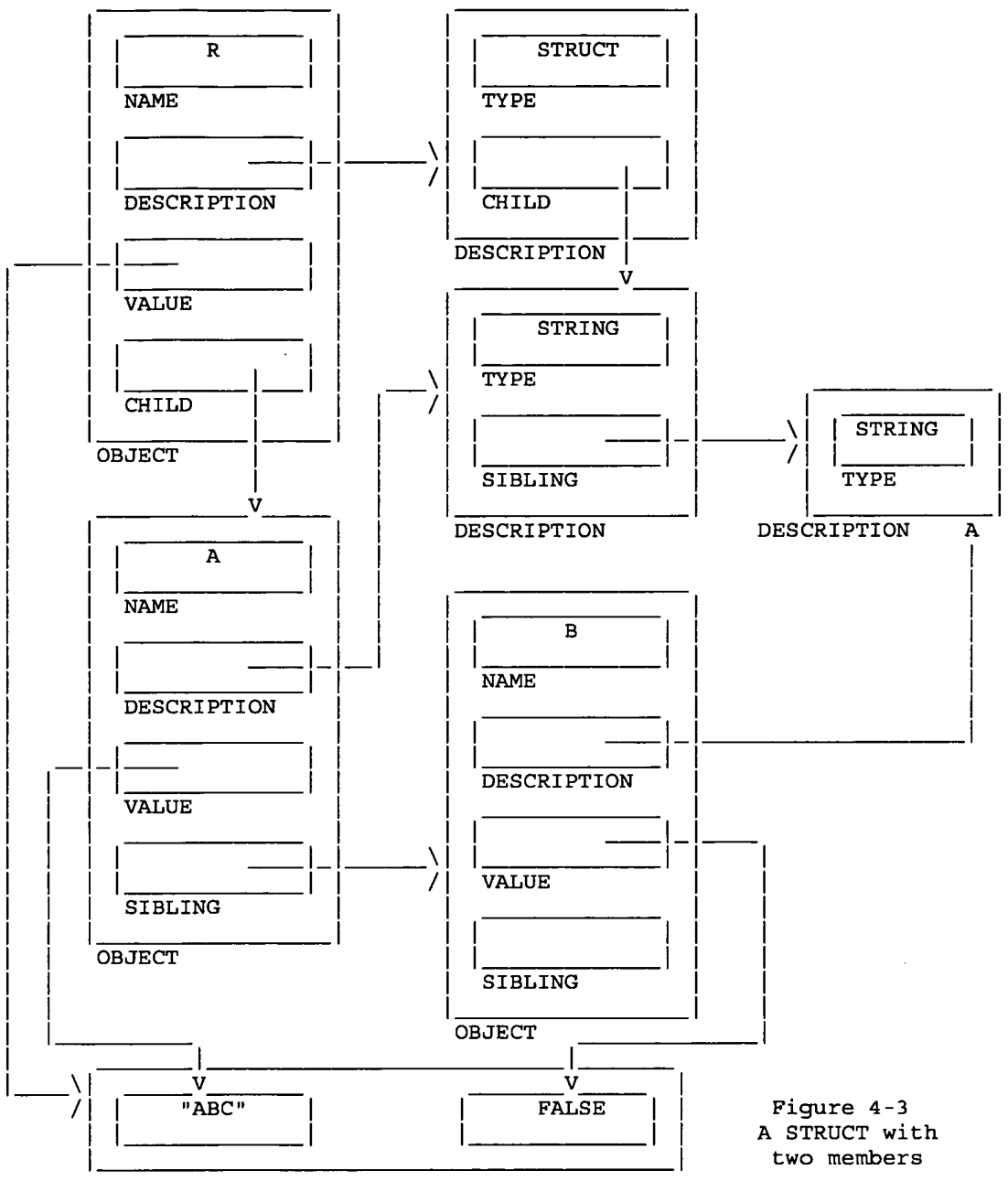


Figure 4-3
A STRUCT with
two members

Figure 4-4 shows a list named L. Here a similar structure of objects is implied, but because of the regularity of the structure, not all the boxes labeled 'object' are actually present.

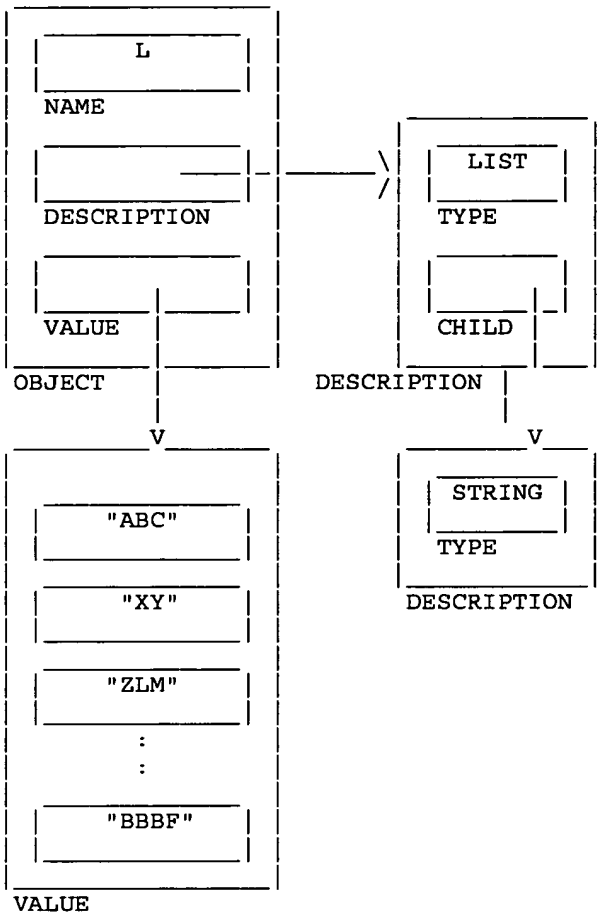


Figure 4-4
A LIST

L has a variable number of components, all satisfying the description subordinate to L's description.

We could imagine an 'object' box for each string in L. Each of these boxes would point to its respective string and to the common description of these strings. Instead, we think in terms of creating such boxes as we need them.

4.5 Definitions of types

Following are some more precise definitions of types, in terms of the present model. These serve the purpose of establishing more firmly the semantics of our structure of objects, descriptions and values; however, they should not be thought of as providing a definition for the completed language specification.

An object of type STRING has a value which is a sequence of characters (figure 4-1).

An object of type BOOL has a value which is a truth value (TRUE or FALSE -- figure 4-1).

An object of type DIR has subordinate objects, each having its own description and value. Subordinate objects can be added and deleted at will (figure 4-2).

An object of type STRUCT has subordinate objects, each of which has a description which is subordinate to the STRUCT's description, and a value contained in the STRUCT's value. The number, order and description of components is fixed when the STRUCT is created (figure 4-3).

An object of type LIST may be thought of as having imaginary subordinate objects, whose existence is simulated by the use of appropriate techniques in processing the LIST. Each of these has the same description, which is subordinate to the description of the LIST. Each has a distinct value, contained in the value of the LIST. In fact, only the LIST object, the LIST and component descriptions, and the values exist (figure 4-4).

An object of type DESC has a description as its value. This value is the same sort of entity which serves as the description of other objects.

An object of type FUNC has a function call as its value. We will be able to say more about this after functions have been discussed.

An object of type OPD has an operation descriptor as its value. (see 4.7 for details).

4.6 Object environment

There are three categories of objects in the model datacomputer. These are p/objects, t/objects, and i/objects.

P/objects are permanent objects created explicitly with language functions. They correspond to the idea of stored data in the real datacomputer. There are three special objects. These are special only in that they are created as part of initializing the environment, rather than as the result of executing a language function. These are named STAR, BLOCK and TOP/LEVEL. All three are of type DIR.

An object is a p/object if it is subordinate to STAR; it is a t/object if it is subordinate to BLOCK. TOP/LEVEL is subordinate to BLOCK. (see figures 4-5 and 4-6).

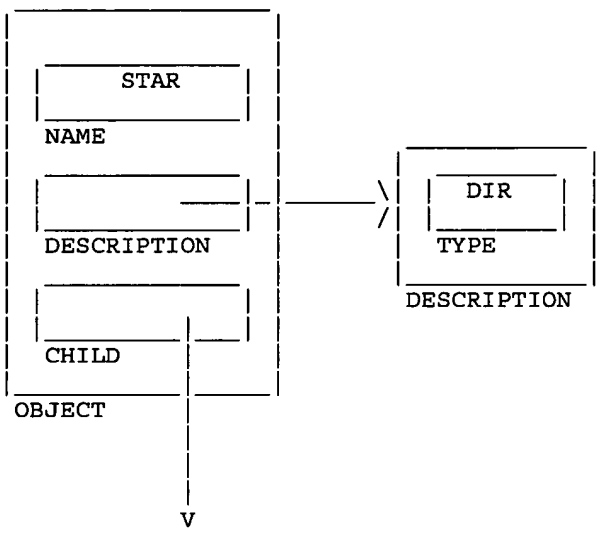


Figure 4-5
STAR and p/objects

T/objects are temporary objects, also created explicitly with language functions. However, these correspond to user-defined temporaries, both local to requests and "top-level" (i.e. not local to any request, but existing until deletion or logout.)

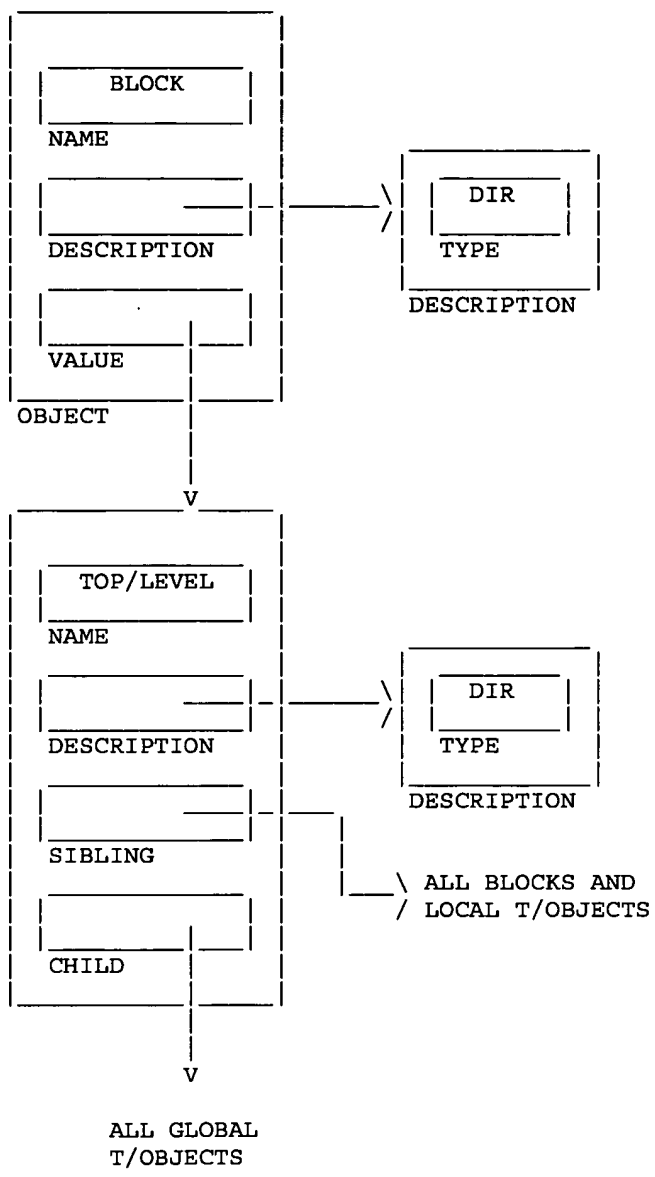


Figure 4-6
BLOCK, TOP/LEVEL and t/objects

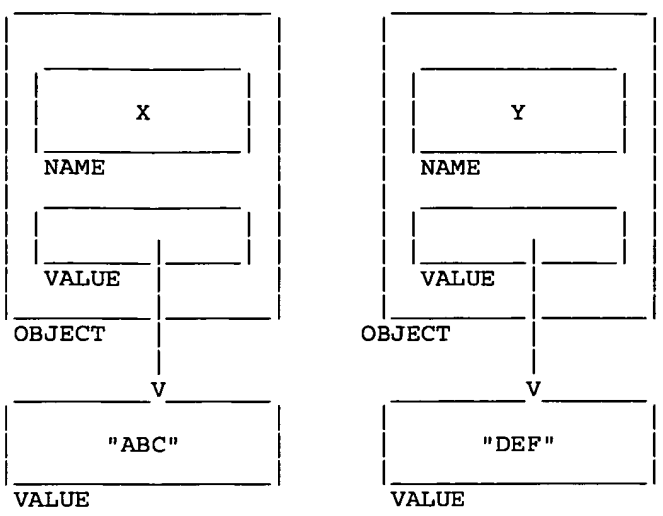
I/objects are internal, system-defined objects whose creation and deletion is implicit in the execution of some language function.

I/objects are hung directly off of function calls (objects of type FUNC), and are always local to the execution of such function calls. They correspond to the notions of (1) literal, and (2) compiler- or interpreter-generated temporary.

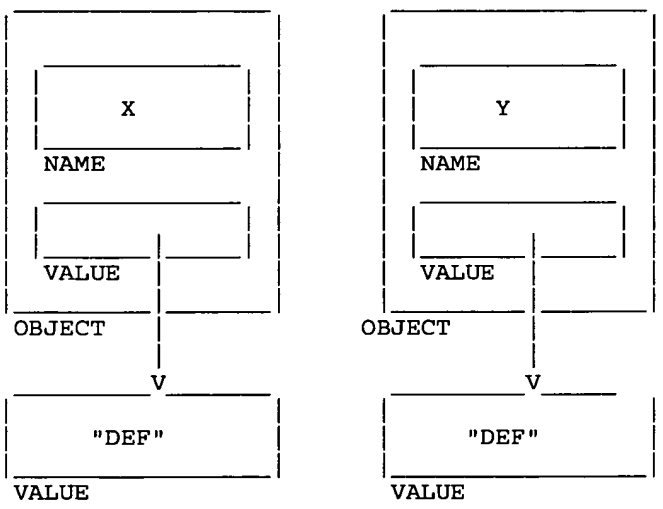
4.7 Primitive Language Functions

Here we discuss the primitive language functions presently implemented in the model and likely to be of most interest. In this section, the emphasis is on relating functions to one another. Section 4.8 contains more detail and examples.

Assign operates on a pair of objects, called the target and the source. The value of the source is copied into the value of the target. Figure 4-7 shows a pair of objects, X and Y, before and after execution of an assignment having X as target and Y as source. Presently, assignment is defined only for objects of type BOOL and objects of type STRING. The objects involved must have identical descriptions.



BEFORE ASSIGNMENT



AFTER ASSIGNMENT

Figure 4-7
Effect of assignment

A class of primitive functions for manipulating LISTs is defined. These are called _listops_. All listops input a special object called an _operation_descriptor_ or OPD.

To accomplish a complete operation on a LIST, a sequence of listops must be executed. There are semantic restrictions on the composition of such sequences, and it is best to think of the entire sequence as one large operation. The state of such an operation is maintained in the OPD.

Refer back to figure 4-4. There is one box labeled "object" in this picture; this box represents the list as a whole. To operate on any given member we need an object box to represent that member. Figure 4-8 shows the structure with an additional object box; the new box represents one member at any given moment. Its value is one of the components of the LIST value; its description is subordinate to the LIST description. In 4-8, the name of this object is M.

In 4-8 we have enough structure to provide a description and value for M, and this is sufficient to permit the execution of operations on M as an item. However, there is no direct link between object M and object L. The structure is completed by the addition of an OPD, shown in figure 4-9.

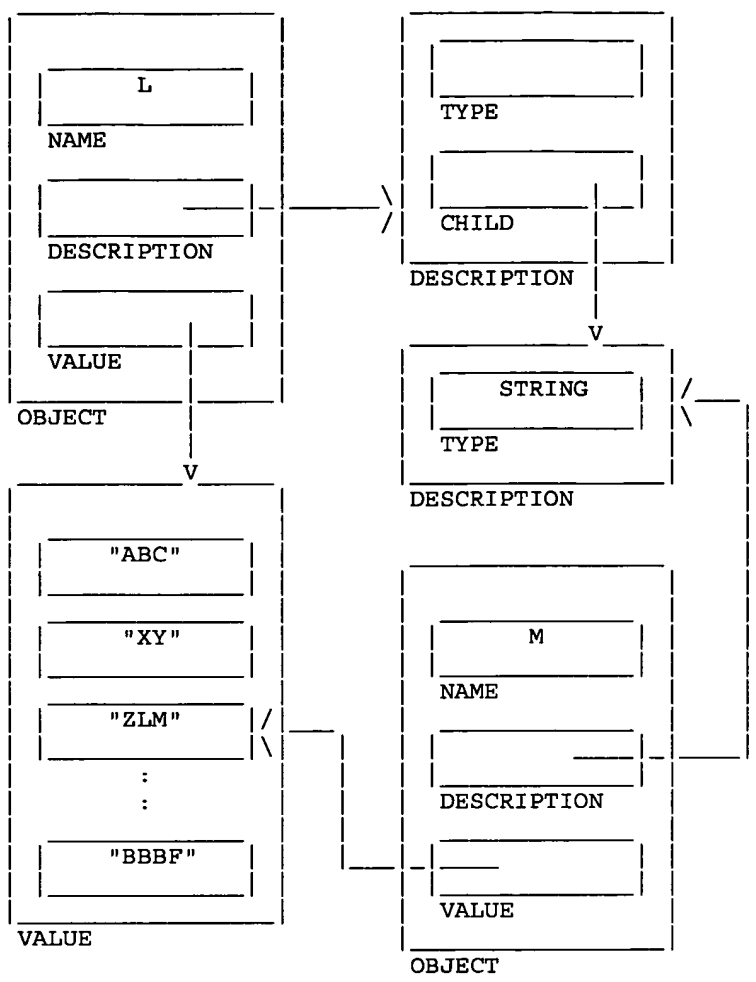


Figure 4-8
LIST and member objects

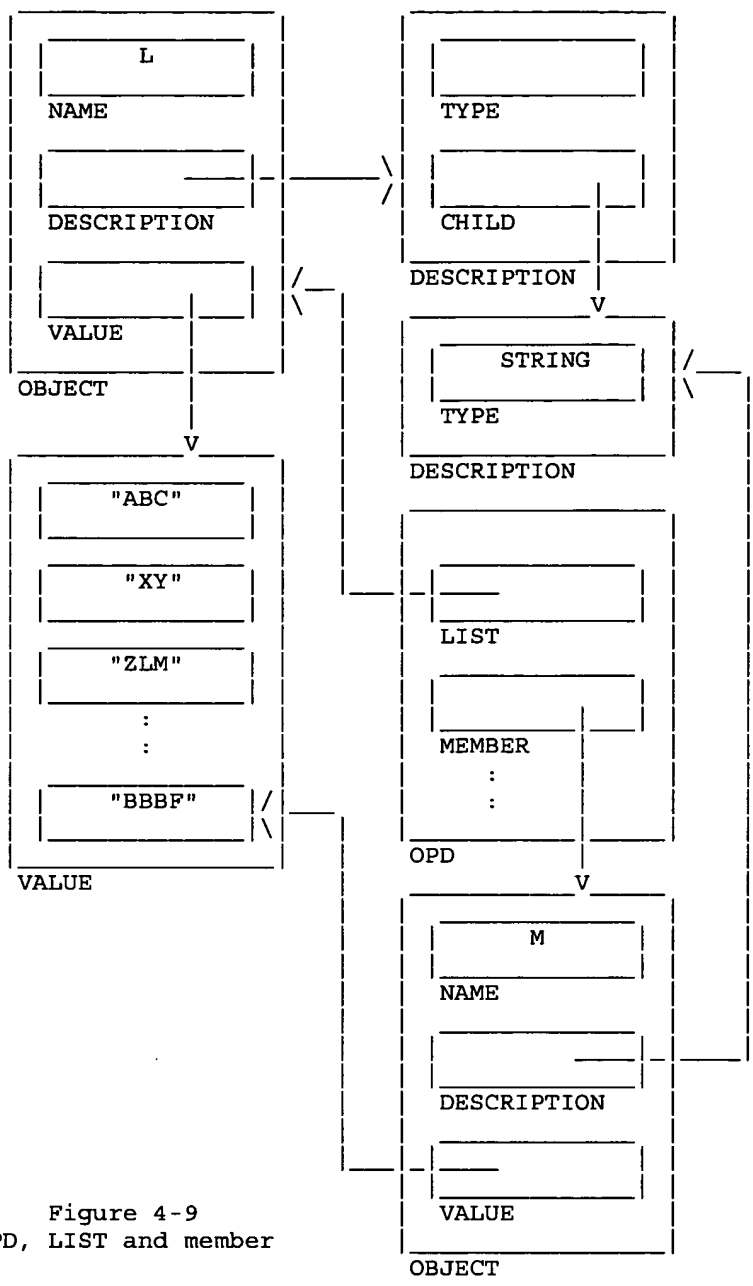


Figure 4-9
OPD, LIST and member

The OPD establishes the object relationship, and contains information about the sequence of primitive listops in progress. When sufficient information is maintained in the OPD, we have in 4-9 a structure which is adequate for the maintenance of the integrity of the LIST and of the global list operation. In addition to LIST and member pointers, the OPD contains information indicating: (1) which suboperations are enabled for the sequence, (2) the current suboperation, (3) the instance number of the current LIST member, (4) an end-of-list indicator. The suboperations are add/member, delete/member, change/member and get/member. All apply to the current member. Only suboperations which have been enabled at the beginning of a sequence may be executed during that sequence; eventually, the advance knowledge of intentions that is implied by this will provide important information for concurrency control and optimization.

Presently, an OPD relates a single member object to a single LIST object. This imposes an important restriction on the class of operation sequences which can be expressed. Any LIST transformation requiring simultaneous access to more than one member must be represented as more than one sequence. (And we do not yet solve the problems implied in concurrent execution of such sequences, even when both are controlled by one process.)

Any transformation of a LIST can still be achieved by storing intermediate results in temporary objects; however, it is certainly more desirable to incorporate the idea of multiple current members into the semantics of the language, than it is to use such temporaries. An important future extension of the listops will deal with this problem.

There are six listops: listop/begin, listop/end, which/member, end/of/list, open/member and close/member.

Listop/begin and listop/end perform the obvious functions of beginning and terminating a sequence of listops. Listop/begin inputs LIST and member objects, an OPD, and a specification of suboperations to enable. It initializes the OPD, including establishment of the links to LIST and MEMBER objects. After the OPD-LIST-member relationship has been established, it is only necessary to supply the OPD and auxiliary parameters as input to a listop in the sequence. From the OPD everything else can be derived.

Listop/end clears the OPD and frees any resources acquired by listop/begin.

Which/member establishes the current member for any suboperations. This is either the first LIST member, the last LIST member, or the next LIST member. This listop merely identifies which member is to be operated on; it does not make the contents of the member accessible.

Open/member and close/member bracket a suboperation. The suboperation is indicated as an argument to open/member. Open/member always establishes a pointer from the member object to the member value; close/member always clears this pointer. In addition, each of these listops may take some action, depending on the suboperation.

The details of the action would be dependent on the representation of the LIST in storage, the size of a LIST member, and choices made in implementation.

Between execution of the open/member and the close/member, the data is accessible. It can always be read; in the case of the add/member and change/member suboperations, it can also be written into.

End/of/list tests a flag in the OPD and returns an object of type BOOL. The value of the object is the same as the value of the flag; it is TRUE if a get/member, change/member or delete/member would be unsuccessful due to a which/member having moved "beyond the end". T his listop is provided so that it is possible to write procedures which terminate conditionally when all members have been processed.

Get/struct/member provides the ability to handle STRUCTs. Given a STRUCT object which points to the STRUCT value, it will establish a pointer from a given member object to the member value. (The pointer it establishes is represented by a dashed line in figure 4-10).

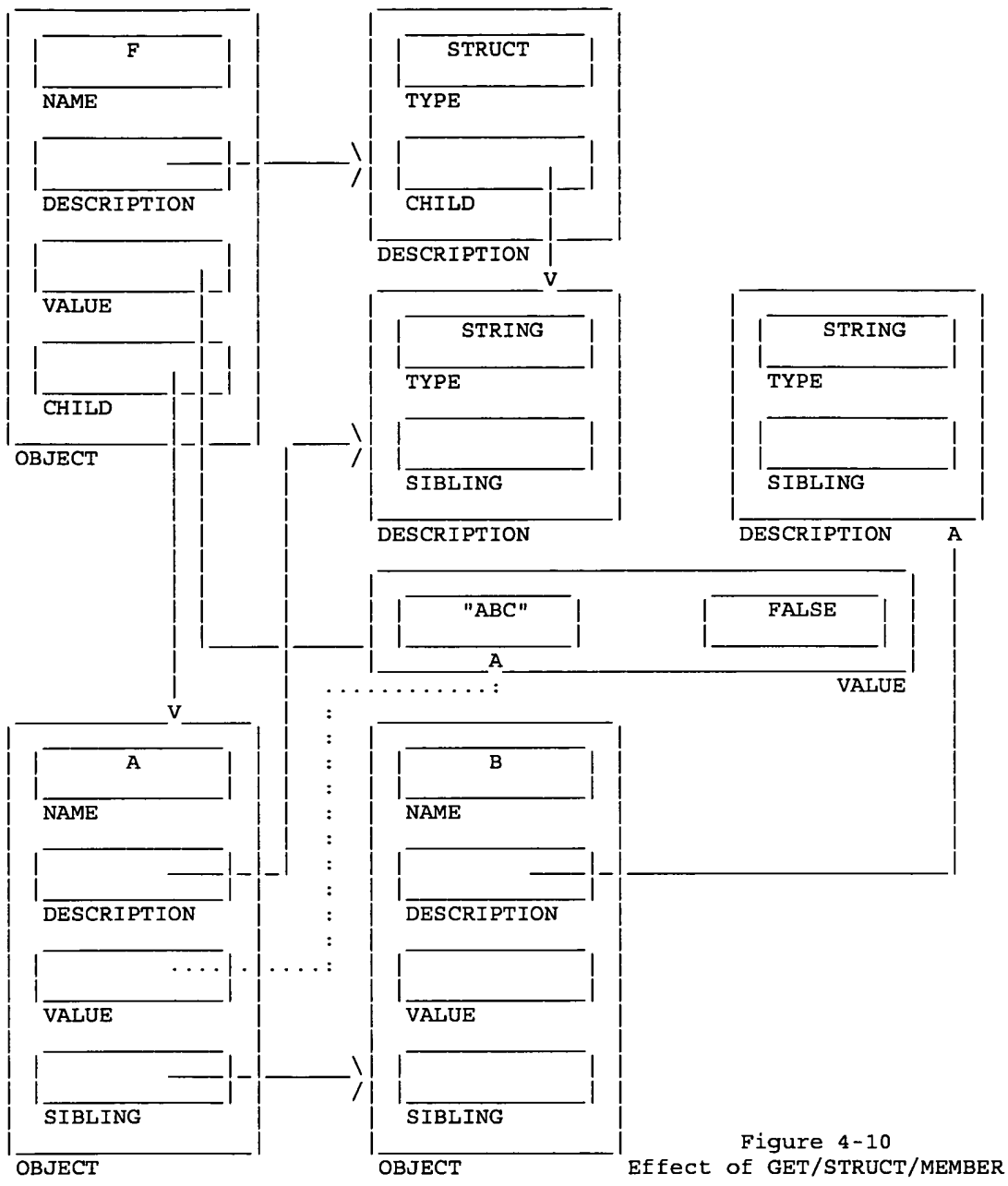


Figure 4-10
Effect of GET/STRUCT/MEMBER

The primitives discussed so far (assign, listops, and get/struct/member) provide a basic facility for operating on structures of LISTs, STRUCTs and elementary items. Using only them, it is possible to transfer the contents of one hierarchical structure to another, to append structures, to delete portions of structures, and so on. To perform more interesting operations facilities for control and selection are needed.

A rudimentary control facility is provided through the primitives if/then, if/then/else, till and while. All of these evaluate one primitive function call, which must return a BOOL. Based on the value of this BOOL some action is taken.

Let A and B be function calls. If/then(A,B) will execute B if A returns TRUE. If/then/else(A,B,C) will execute B if A returns TRUE; it will execute C if A returns FALSE. The while and till operators iterate, executing first A then B. While terminates the loop when A returns FALSE; till terminates the loop when A returns TRUE. If this happens the first time, B is never executed.

So far, we have mentioned one function which returns a BOOL: the listop, end/of/list. Two other classes of functions which have this property are the booleans and the comparisons. There are 3 primitive booleans (and, or, not) and six primitive comparisons (equal, less/than, greater/than, not/equal, less/than/or/equal, greater/than/or/equal -- only equal is implemented at time of publication).

The booleans input and output BOOLs; the comparisons input pairs of elementary objects having the same description and output BOOLs. Expressions composed of booleans and comparisons on item contents are one of the principal tools used in selectively referencing data in data management systems.

With the booleans, the comparisons, and the primitives identified earlier, we can perform selective "retrievals". That is, we can transfer to LIST B all items in LIST A having a value of 'ABC'. In fact, we now have a (semantically) general ability to perform content-based retrievals and updates on arbitrary hierarchical structures. We can even program something as complex as the processing of a list of transactions against a master list, which is one of the typical applications in business data processing.

Of course, we would not expect users of datalanguage to express requests at the level of listops. Further, the listops defined here are not a very efficient way of performing some of the tasks we have mentioned. To get good solutions, we need both higher-level operators and other primitives which use other techniques in processing.

In addition to those already discussed, the model contains functions

for: (1) referencing an object by qualified name, (2) generating a constant, (3) generating data descriptions, (4) writing compound functions and blocks with local variables, (5) creating objects.

The facilities for generating constants and data descriptions (which are a special case of constants) are marginal, and have no features of special interest. Obviously, data description will be an important concern in the modeling effort later on.

Object referencing functions permit reference to t/objects and p/objects (these terms are defined in 4.6). A p/object is referenced by giving the pathname from STAR to it. A t/object is referenced by giving the pathname from the block directory in which it is defined to it.

Compound/function permits a sequence of function calls to be treated syntactically as a single call. Thus, for example, in if/then(A,B), B is frequently a call to compound/function, which in turn calls a sequence of other functions.

Create takes two inputs: a superior object and a description. The superior must be a directory. The new object is created as the leftmost child of the directory; its name is determined by the description.

4.8 Details of primitive language functions

This section provides specifications for the primitives discussed in the previous section. We are still omitting details when we judge them to be of no general interest; the objective is to provide enough information for the reader to examine examples.

Most of the primitives occur at two levels in the model. The internal primitives are called i/functions and the external, or language primitives are called l/functions. The relationship between the two types are explained in 4.9. In this section we discuss i/functions.

L/functions input and output _forms_, which are tree structures whose terminal nodes are atoms. The atoms are such things as function names, object names, literal string constants, truth values and delimiters. Calls to i/functions are also expressed as forms.

Any form can be evaluated, yielding some object. A form which is an i/function call yields the value returned by the i/function: another form. In general, the form returned by an i/function call will, when evaluated, yield a datalanguage object (that is, the sort of object we have been represented by an "object box" in the drawings).

4.8.1 Name recognition functions

These return a form which evaluates to an object.

L/TOBJ

Input must name a temporary object subordinate either to TOP/LEVEL or a block directory.

L/POBJ

Input must name a permanent object (i.e., an object subordinate to STAR).

Typical calls are L/POBJ(X.Y.Z) and L/TOBJ(A).

4.8.2 Constant generators

Each of these inputs an atomic symbol yielding a value for a constant to be created. Each returns a form which will evaluate to an object having the specified value and an appropriate description.

LC/STRING - a typical call is LC/STRING('ABC')

LC/BOOL - a typical call is LC/BOOL(TRUE)

4.8.3 Elementary item functions

These input and output forms evaluating to elementary objects (objects which can have no subordinate object -- in effect, objects whose value is regarded as atomic). Eventually all the comparison operators will be implemented.

L/ASSIGN

Inputs must evaluate either to STRINGs or BOOLs. Outputs a form which transfers the value of the second to the first. Typical call:

```
    L/ASSIGN(L/TOBJ(A),LC/STRING('XYZ'))
```

The output form, when evaluated, will copy 'XYZ' into A's value.

L/EQUAL

Inputs a pair of forms evaluating to objects, which must have identical descriptions and be BOOLs or STRINGs. Returns a form evaluating to an object of type BOOL. Value of this object is TRUE if inputs have identical descriptions and values; otherwise it is false. Typical call:

```
    L/EQUAL(L/TOBJ(X),LC/STRING('DEF'))
```

L/AND, L/OR, L/NOT

The standard boolean operators. Inputs are forms evaluating to BOOLs; output is a form evaluating to a BOOL. L/AND and L/OR take two inputs; L/NOT one. Typical call:

```
    L/AND( L/EQUAL(L/TOBJ(X),LC/STRING('DEF')),
                   L/EQUAL(T/TOBJ(Y),LC/STRING('GHI')) )
```

The form returned will, when evaluated, return TRUE if both X has value 'DEF' and Y has value 'GHI'.

4.8.4 Data description functions

These all return a form evaluating to a description (i.e. that which is represented in our drawings by a box labeled "description").

LD/STRING

Inputs 3 parameters specifying the name, size option and size for the string. Typical call:

```
    LD/STRING(X,FIXED,3)
```

This call returns a form evaluating to a description for a fixed-length 3-character string named X.

LD/LIST

Inputs two forms. The first is the name of the LIST and the second evaluates to a description of the LIST member. Typical call:

```
    LD/LIST(L,LD/STRING(M,FIXED,3))
```

Creates the structure shown in figure 4-11, and returns a form evaluating to the description represented by the upper box.

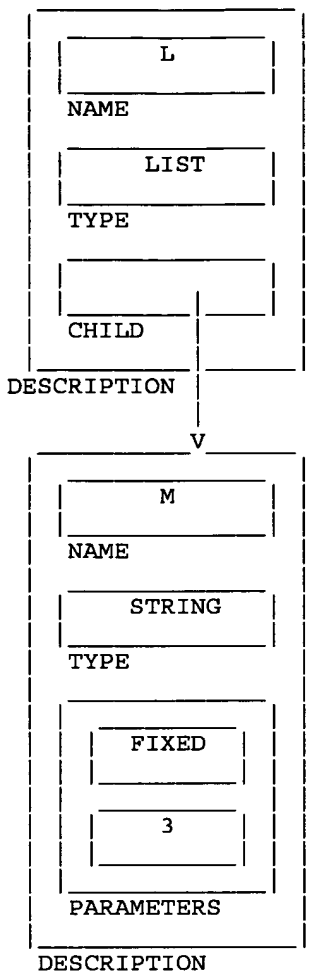


Figure 4-11
LIST and member descriptions

LD/STRUCT

Inputs a form to use as the name for the STRUCT and one or more forms evaluating to descriptions; these are taken as the descriptions of the members. Typical call:

```
    LD/STRUCT(R,
            LD/STRING(A,FIXED,3)
            LD/BOOL(B) )
```

produces the structure shown in 4-12; returns a form evaluating to the top box.

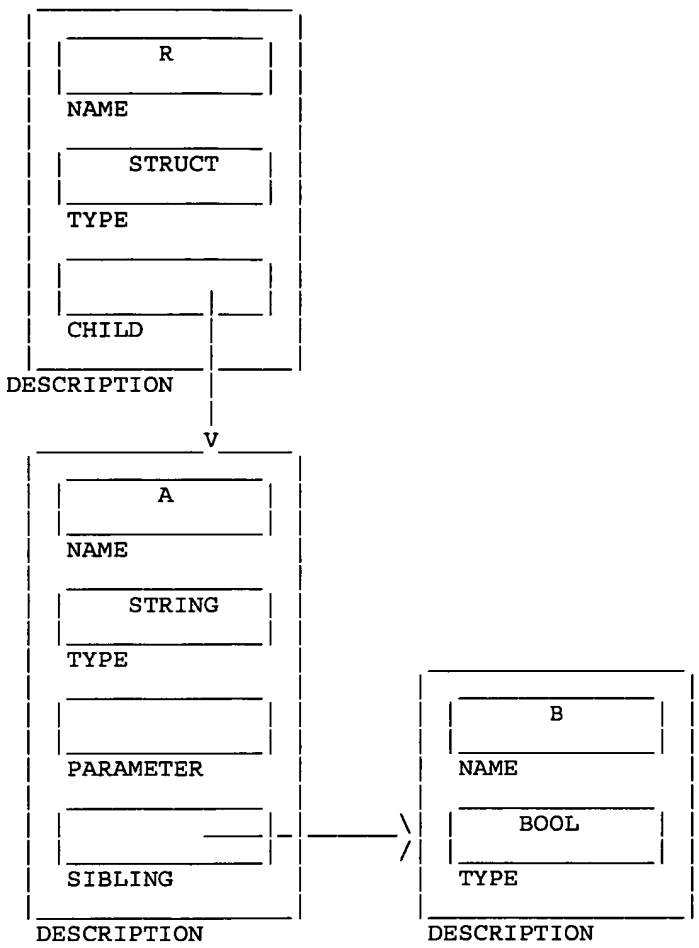


Figure 4-12
STRUCT and member descriptions

LD/BOOL, LB/DIR, LD/OPD, LD/FUNC, LD/DESC

Each inputs a name and produces a single description; each returns a form evaluating to the description produced. Typical call:

```
LD/BOOL(X)
```

4.8.5 Data creation

L/CREATE

Inputs two forms and evaluates them. First must yield an object of type DIR; second must yield a description for the object to be created. Creates the object and returns a form, which, when evaluated, will generate a value for the new object. A simple example:

```
L/CREATE(L/TOBJ(X),LD/BOOL(Y))
```

Figure 4-13 shows the directory X before execution of the above call. It contains only an OPD. After execution, the directory appears as in 4-14. Creation of a value for Y occurs when the form returned by L/CREATE is evaluated (covered in section 4.9).

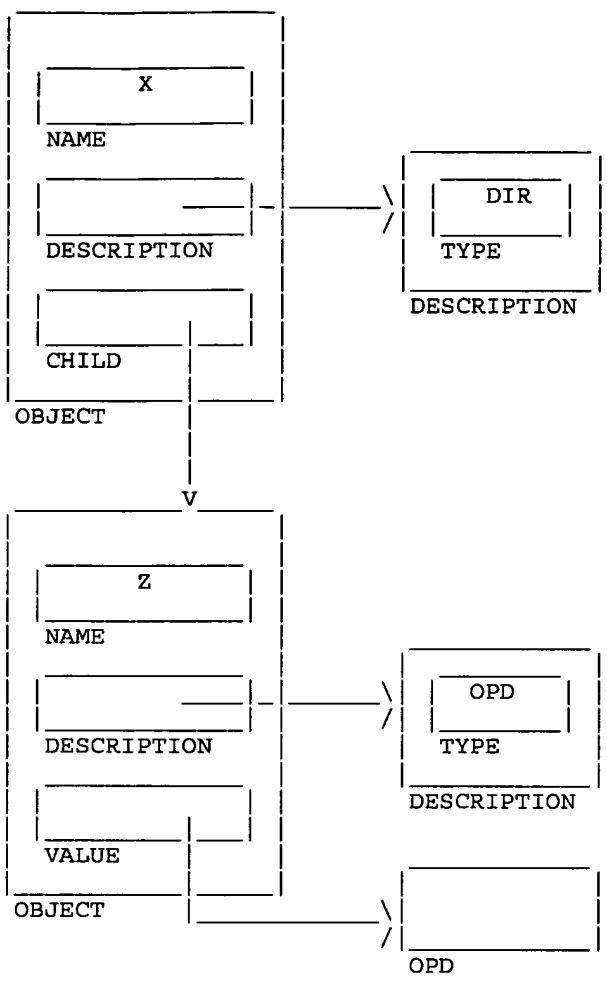


Figure 4-13
X and Z before creation of Y

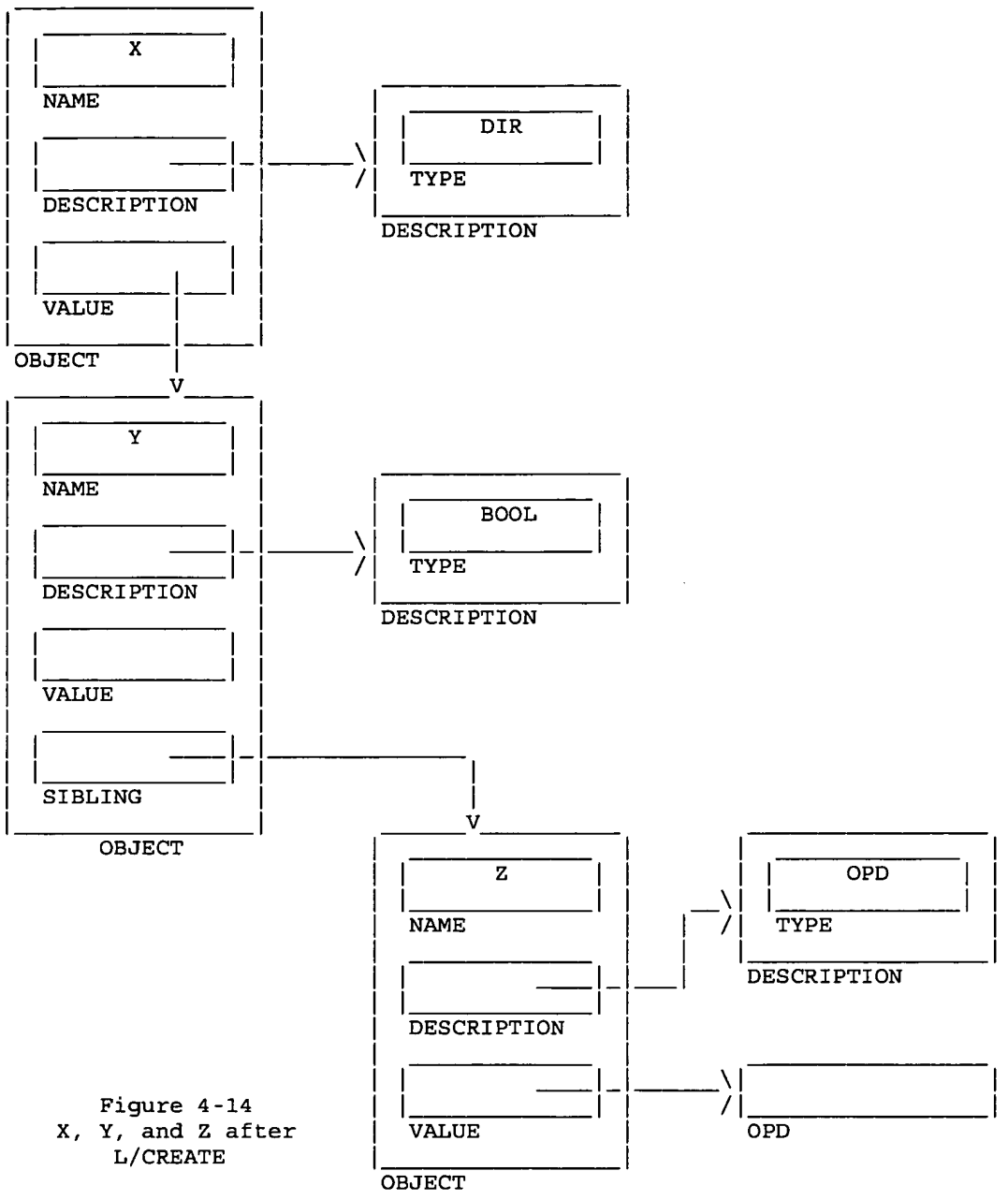


Figure 4-14
X, Y, and Z after
L/CREATE

4.8.6 Control

L/IF/THEN, L/IF/THEN/ELSE

Used to request conditional evaluation of a form. Typical call:

```
L/IF/THEN(L/EQUAL(L/TOBJ(A),LC/STRING('ABC'),
        L/ASSIGN(L/TOBJ(B),LC/STRING('DE')))
```

The form returned will do the following, when evaluated: if A has value 'ABC', then store 'DE' in the value of B.

L/WHILE, L/TILL

These iterate conditionally, as explained in the previous section. Examples appear later.

L/CF

Compound function: it inputs one or more forms and returns a form which, when evaluated, will evaluate each input in sequence. Typical call:

```
L/CF(L/ASSIGN(L/TOBJ(R.A),LC/STRING('XX')),
        L/ASSIGN(L/TOBJ(R.B),LC/STRING('YY')))
```

When the output of L/CF is evaluated, it will assign new values to R.A and R.B.

4.8.7 Listops

These primitives are executed in sequences in order to perform operations on LISTs. With the exception of L/END/OF/LIST these functions output forms which are evaluated for effect only; that is, the output forms do not themselves return values.

L/LISTOP/BEGIN

Inputs forms evaluating to: (1) a LIST, (2) an object to represent the current LIST member, (3) an OPD. Also, inputs a list of atomic forms whose values are taken as suboperations to enable. Typical call:

```
L/LISTOP/BEGIN(L/POBJ(F),L/TOBJ(R),
                L/TOBJ(OPF),ADD,DELETE)
```

This returns a form that will initialize a sequence of listops to be performed on F. Caller has previously created R and OPF. He intends to ADD and DELETE list members.

All subsequent calls in this sequence of listops need specify only the OPD and auxiliary parameters.

L/LISTOP/END

Inputs a form evaluating to an OPD. Outputs a form which, when evaluated, clears OPD and breaks relationships between OPD, LIST and member objects.

L/WHICH/MEMBER

Inputs two forms. First evaluates to an OPD; second is one of FIRST, LAST, NEXT. The form output, when evaluated, will establish a new current member for the next suboperation. Note: this does not make the value of the member accessible, it simply identifies it by setting the instance number in the OPD. A typical call:

```
L/WHICH/MEMBER(L/TOBJ(OPF),NEXT)
```

When a which/member causes advance beyond the end of the list, a flag is set in the OPD.

L/END/OF/LIST

Inputs a form evaluating to an OPD. Outputs a form which, when evaluated, returns a BOOL. This has value TRUE if the end of list flag in the OPD is on.

L/OPEN/MEMBER

Inputs a form evaluating to an OPD and a form which must be one of ADD, DELETE, GET, CHANGE. Outputs a form which, when evaluated, will initiate the requested suboperation on the current LIST member. The suboperation always establishes the pointer from the member object to the current member value instance. In addition, in the case of ADD this value must be created. Typical call:

```
L/OPEN/MEMBER (L/TOBJ (OPF) ,ADD)
```

L/CLOSE/MEMBER

Inputs a form evaluating to an OPD. Outputs a form which, when evaluated, will complete the suboperation in progress. A typical call:

```
L/CLOSE/MEMBER(L/TOBJ(OPF))
```

Always clears the pointer from member object to member value. In addition, in the case of DELETE, removes the member value from the LIST. In the case of ADD enters the member value in the LIST. Makes the member added the current member, so that a sequence of ADDs executed without intervening which/members will add the new members in sequence.

An elaborate example, involving listops and several other primitives, appears in section 4.10.

4.9 Execution cycle

The model datacomputer has a two-part execution cycle: it first compiles requests, then interprets them. A "request" is an l/function call; "compilation" is the aggregate result of executing all the l/function calls involved in the request (typically this is many calls, as there are usually several levels of nested calls, with the results of the inner calls being delivered as arguments to the next level of calls). Usually, the process of executing an l/function involves a simple macro expansion, preceded by some binding, checking and (eventually) optimization.

The compiled form consists wholly of atomic symbols and i/function calls. The i/functions are internal primitives which input and output datalanguage objects (the entities represented by the boxes labeled "object" in the drawings).

Each of the l/functions discussed compiles into a single i/function; thus the macro expansion aspect of compilation is presently trivial. However, this will not be true in general; it is only that these are _primitive_ l/functions that makes it true now.

The decision to use a compile-and-interpret cycle calls for some explanation. The way to understand this, is to think in terms of the functions that would be performed in a strictly interpretive system. There would still be a requirement to perform global checks on the validity of the request in advance of the execution of any part of it. This is because partial execution of an incorrect request can leave a database in an inconsistent state; if this is a large or complex database, the cost of recovery will be considerable. Thus it pays to do as much checking as is possible; when the system is fully developed, this will include a certain element of simple prediction of execution flow; in any case, much more than syntactic checking is implied.

Since any such global checks will be performed in advance of actual execution, they are effectively not part of the execution itself, for any given form. By performing them as part of a separate compilation process, we only formalize a modularity which already effectively exists.

There will still be cases, however, in which checking, binding and optimization functions must be executed during interpretation, if at all. This will occur when the information needed is not available until some of the data has been accessed. When practical, we will provide for such occurrences by designing most functions so that they can be executed as part of either "half" of the cycle.

As the model develops, we expect to get a better feel for this problem;

it is certainly reasonable to end up with a structure in which there are many cycles of compilation and interpretation, perhaps forming a structure in which nesting of cycles within cycles occurs.

4.10 Examples of operations on LISTs

Here we develop an example of an operation on a LIST using primitive l/functions. We first show the function calls required to create a LIST named F and give it a few member values. We then selectively copy certain members to a second LIST G.

To create F:

```
L/CREATE("STAR",LD/LIST(F,
                        LD/STRUCT(R,
                                LD/STRING(A,FIXED,2),
                                LD/STRING(B,FIXED,2))))
```

This creates F as a member of the permanent directory STAR (see section 4.6 for details about STAR). The symbol STAR has a special status in the "language", in that it is one of the few atomic symbols to evaluate directly to an object. (Recall that most permanent objects are referenced through a call to L/POBJ; reserving the symbol STAR is equivalent to reserving STAR as a name and writing L/POBJ(STAR). The solution we choose here is easier to write.) Execution of this call builds the structure shown in 4-15 (except for STAR, which existed in advance of the call). The value initially created for F is an empty LIST--a LIST of zero members.

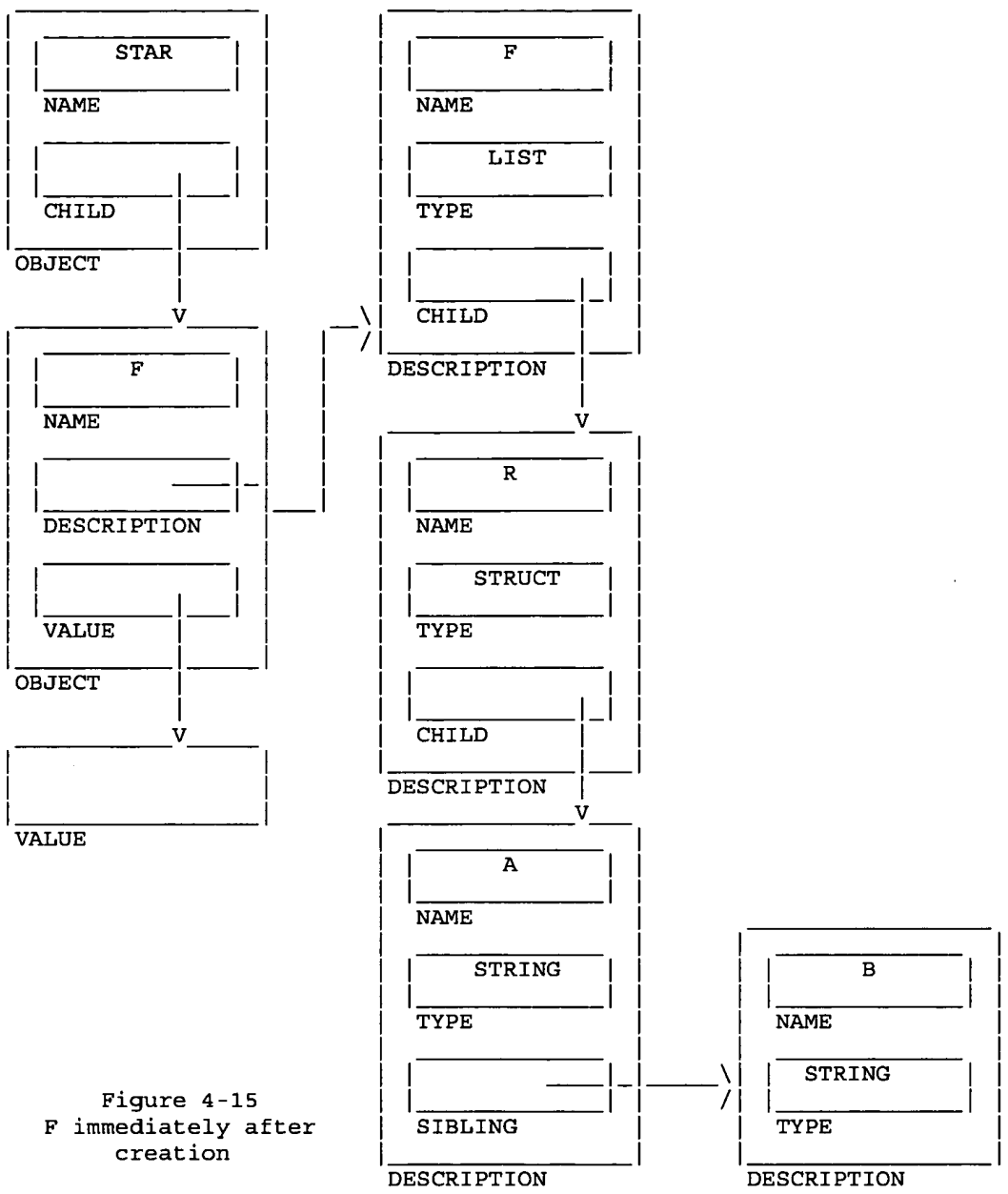


Figure 4-15
F immediately after creation

To add members to F, we need to use listops, and for this we must create two more objects: an object to represent the current member and an operation descriptor (OPD). These are temporaries rather than permanent objects; they are also "top level" (i.e., not local to a request). Temporary, top level objects are created as members of the directory TOP/LEVEL. The calls to create them are:

```
L/CREATE(L/TOBJ(TOP/LEVEL),
                LD/STRUCT(M,
                        LD/STRING(A,FIXED,2),
                        LD/STRING(B,FIXED,2)))
L/CREATE(L/TOBJ(TOP/LEVEL),LD/OPD(OPF))
```

We create M to represent the current member; its description is the same as the one input for a member of F (see the call which created F). The proper way to accomplish this is with a mechanism which shares the actual LIST member description with M; however, this mechanism does not yet exist in our model.

We now wish to add some data to F; each member will be a STRUCT containing two two-character STRINGs.

To begin the listop sequence:

```
L/LISTOP/BEGIN(L/POBJ(F),L/TOBJ(M),
                        L/TOBJ(OPF),ADD)
```

This call establishes the structure shown in figure 4-16. It initializes the OPD, making it point to F and M and recording that only the ADD suboperation is enabled.

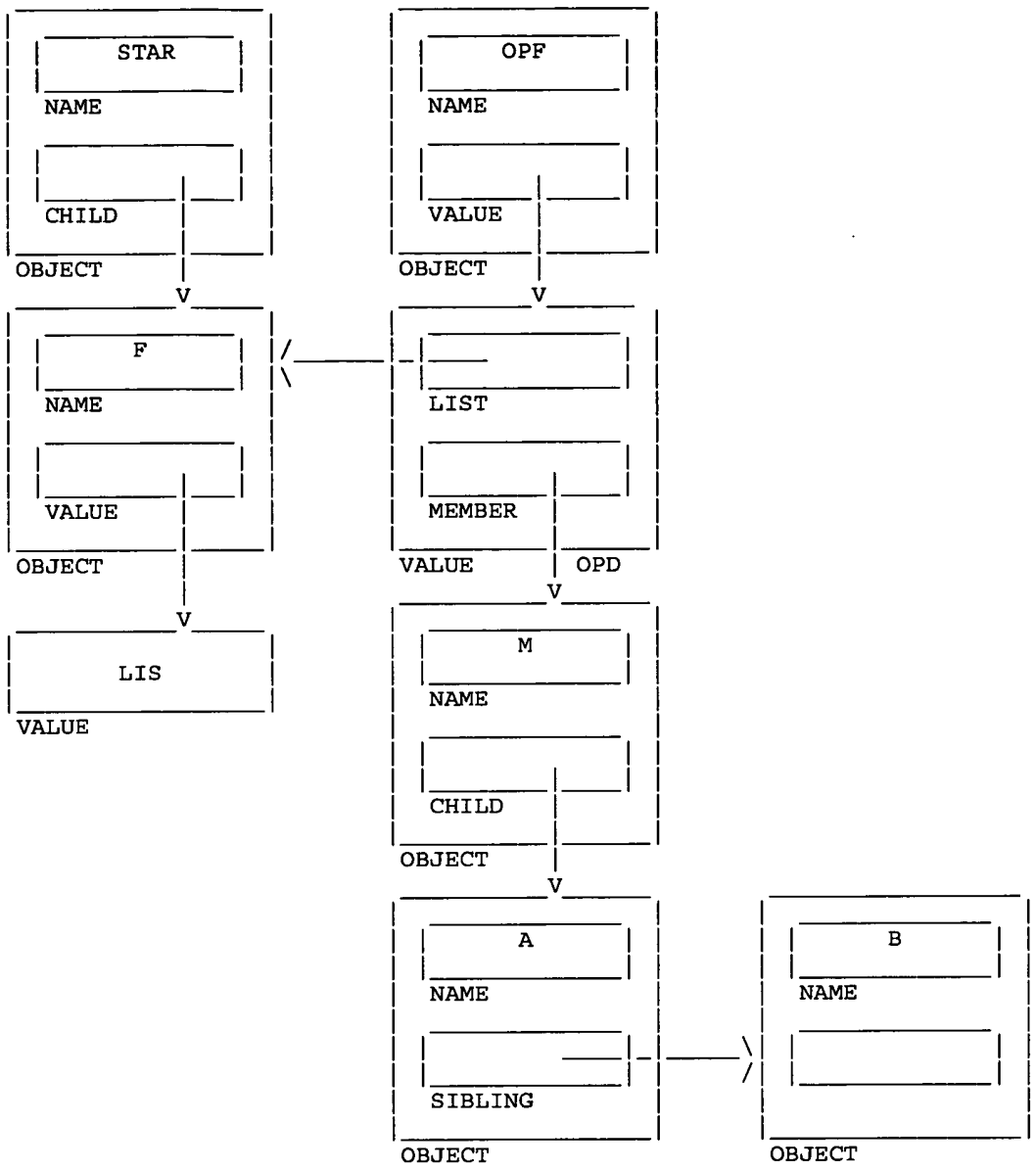


Figure 4-16
F, OPF and M after L/BEGIN/LISTOP

Next we must establish a current member. We want to add members to the end (in this case, adding them anywhere would get the same effect, since the LIST is empty), which is done by making LAST the current member.

```
L/WHICH/MEMBER(L/TOBJ(OP1),LAST)
```

Now, to add a new member to F, we can execute the following:

```
L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('AB'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('CD'))
L/CLOSE/MEMBER(L/TOBJ(OPF))
```

L/OPEN/MEMBER creates a STRUCT value for M. It does not affect the value of F. Each member of the STRUCT value is initialized when the STRUCT is created. The result is shown in 4-17; notice that the STRUCT member values are as yet unrelated to the objects M.A and M.B.

Figure 4-18 shows the changes accomplished by the first L/ASSIGN; the pointer from the object M.A to the value was set up by a GET/STRUCT/MEMBER compiled by L/TOBJ(M.A). The value was filled in by the assign operator. The second assign has similar effect, filling in the second value. The call to L/CLOSE/MEMBER takes the value shown for M in 4-18 (with the second member value filled in) and adds it to the value of F. The result is shown in 4-19; compare with 4-16.

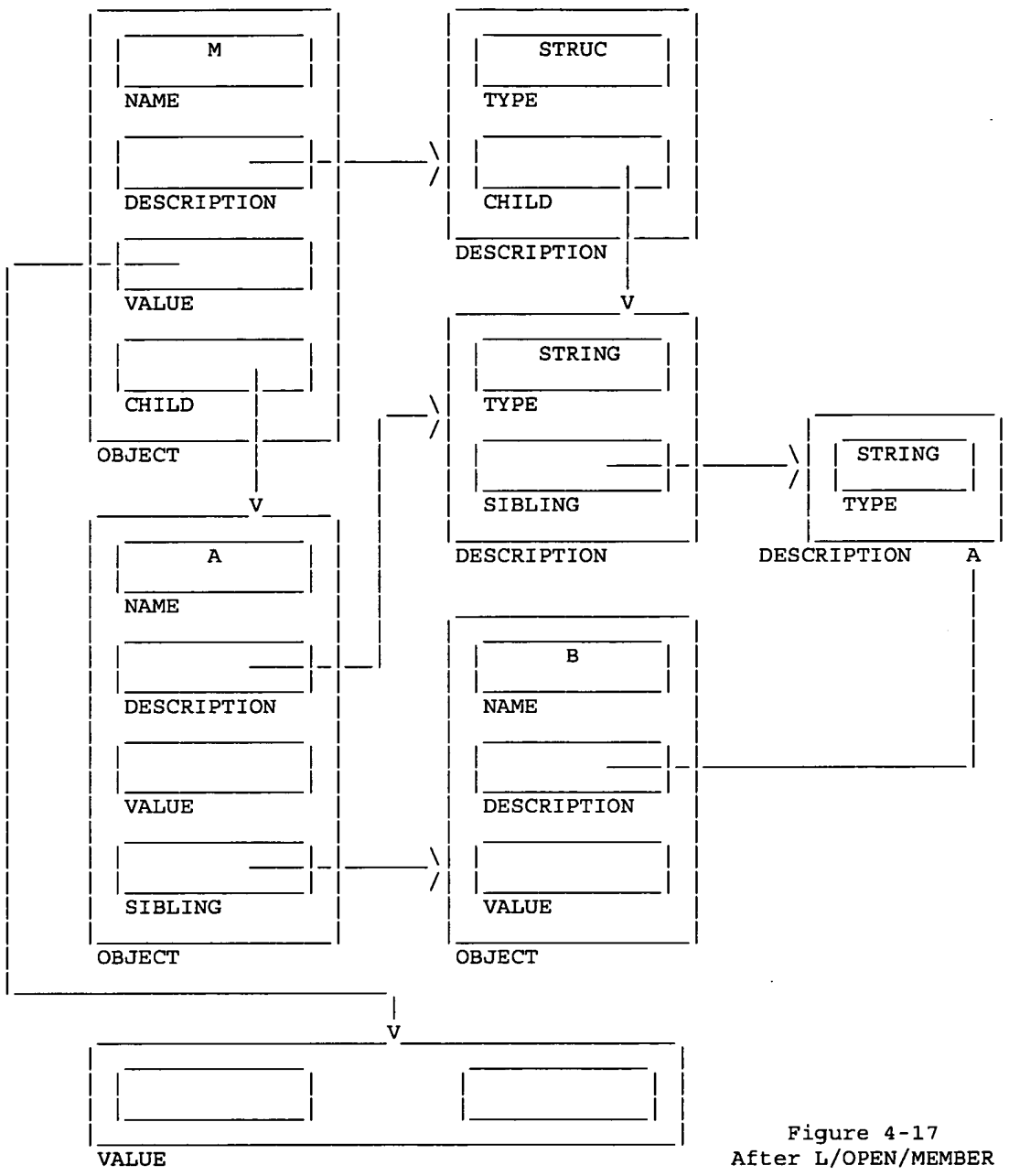


Figure 4-17
After L/OPEN/MEMBER

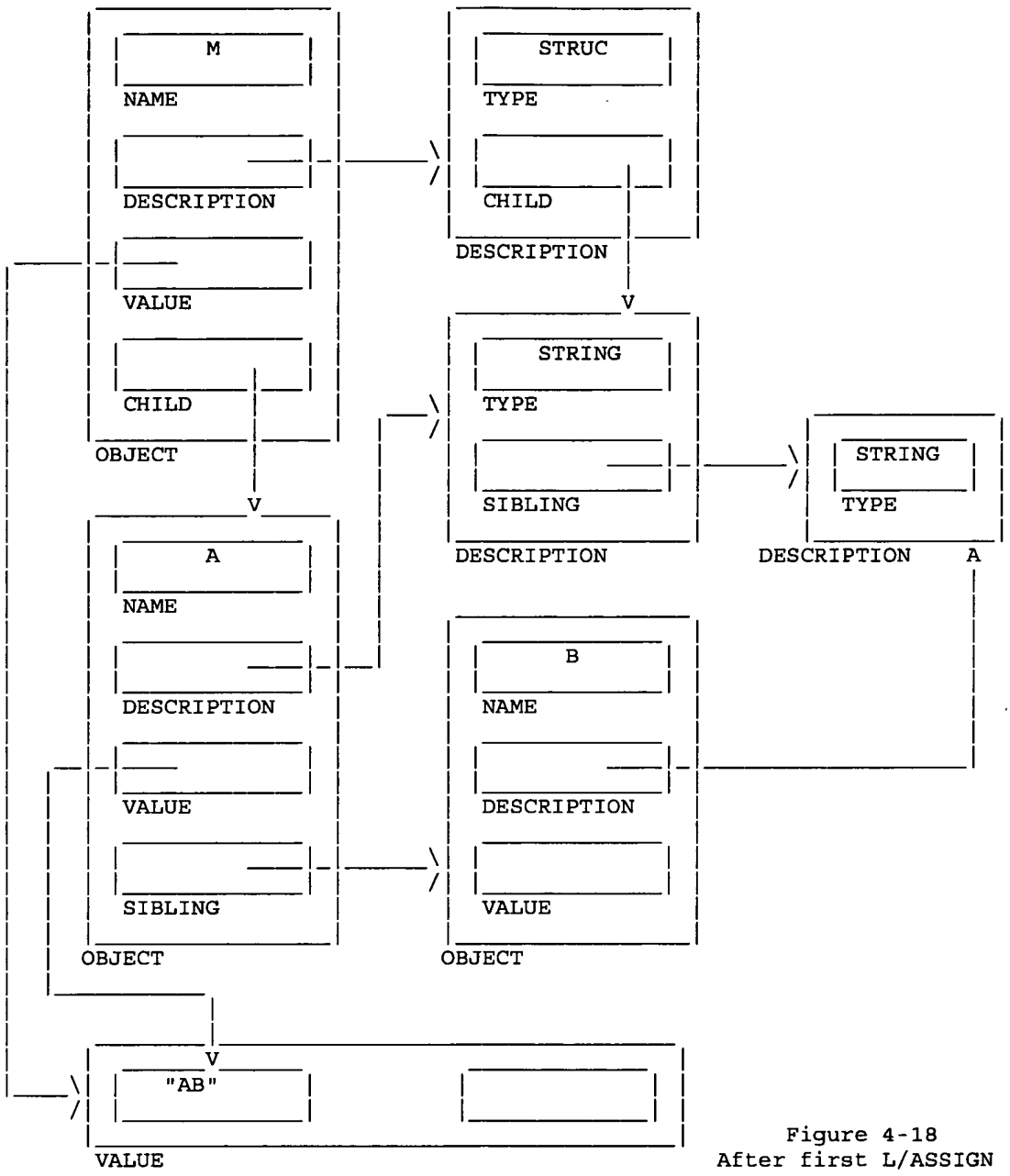


Figure 4-18
After first L/ASSIGN

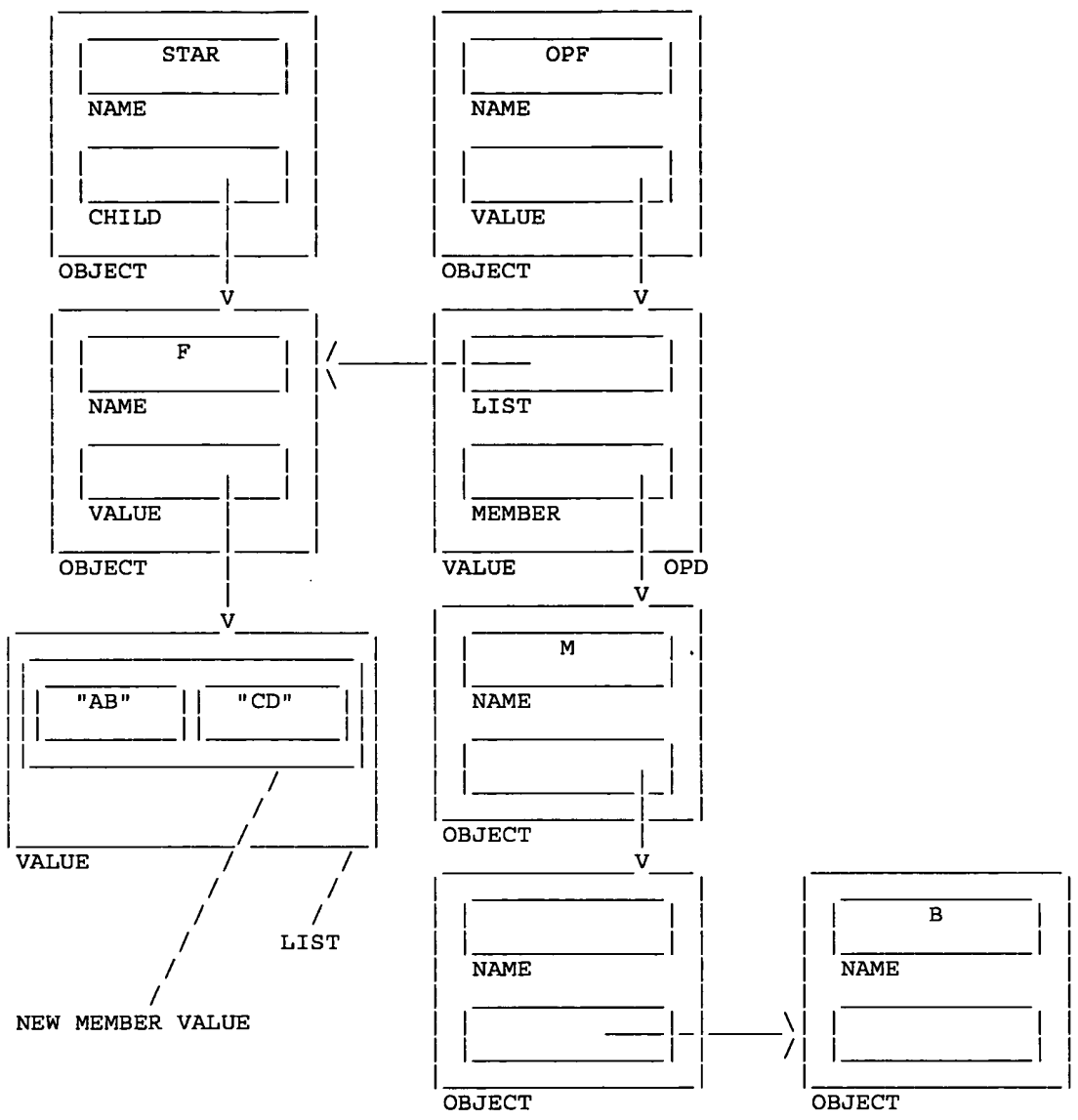


Figure 4-19
After L/CLOSE/MEMBER

By executing similar groups of four primitives, varying only values of constants, we can build up the LIST F shown in 4-20. The calls required are shown below:

```
L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('FF'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('GH'))
L/CLOSE/MEMBER(L/TOBJ(OPF))

L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('AB'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('IJ'))
L/CLOSE/MEMBER(L/TOBJ(OPF))

L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('CD'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('LM'))
L/CLOSE/MEMBER(L/TOBJ(OPF))
```

The add suboperation has the effect of making the member just added, the current member; thus no L/WHICH/MEMBER calls are needed in this sequence.

To terminate the sequence of listops:

```
L/END/LISTOP(L/TOBJ(OPF))
```

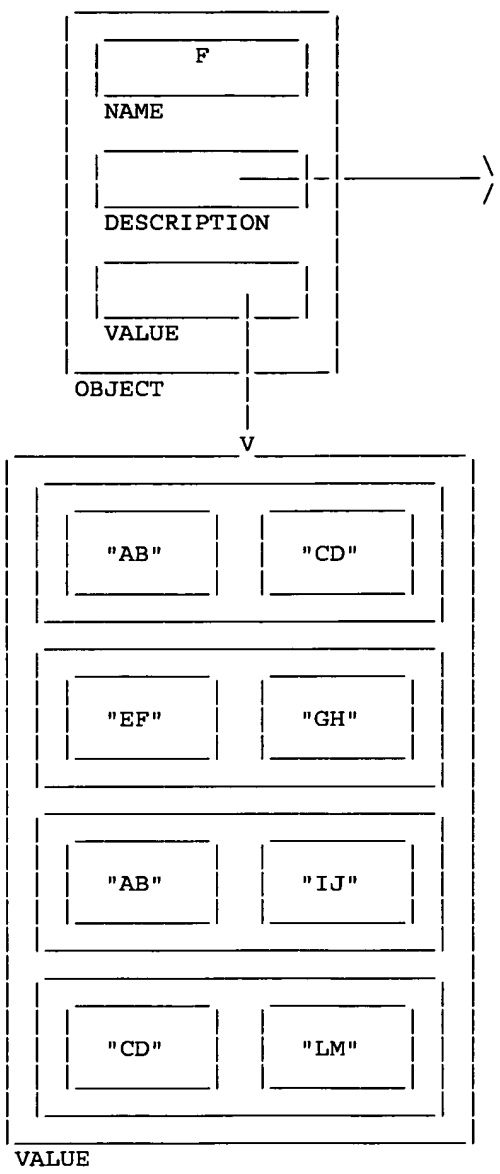


Figure 4-20
After L/END/LISTOP

A slightly more interesting exercise is to construct calls which create a LIST named G, having the same description as F, and then to copy into G all members of F having A equal to 'AB'.

We must first create G, an OPD and an object to represent the current member.

```
L/CREATE("STAR",LD/LIST(G,
                        LD/STRUCT(R,
                                  LD/STRING(A,STRING,2),
                                  LD/STRING(B,STRING,2)))
 L/CREATE(L/TOBJ(TOP/LEVEL),LD/OPD(OPG))
 L/CREATE(L/TOBJ(TOP/LEVEL) ,LD/STRUCT(GM,
                                  LD/STRING(A,STRING,2),
                                  LD/STRING(B,STRING,2)))
```

We now need to initiate two sequences of listops, one on G and one on F.

```
L/BEGIN/LISTOP(L/POBJ(F),L/TOBJ(M),
                   L/TOBJ(OPF),GET)
L/BEGIN/LISTOP(L/POBJ(G),L/TOBJ(GM),
                   L/TOBJ(OPG),ADD)
L/WHICH/MEMBER(L/TOBJ(OPF),FIRST)
L/WHICH/MEMBER(L/TOBJ(OPG),LAST)
```

We will now sequence through the members of F; whenever the current member has A equal to 'AB', we will add a member to G. We then copy the values of the current member of F into the newly added member of G. When the current member does not meet this criterion, we do nothing with it.

First, to write a loop that will execute until we get to the end of F:

```
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),x)
```

Whatever we put in this call to replace "x" will execute repeatedly until the end/of/list flag has been set in OPF.

We must replace "x" with a single function call to in order to give L/TILL what it is looking for. However, we will be executing "x" once for each member of F, and will need to execute several listops each time. The solution is to use L/CF, the compound-function function:

```
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),L/CF(y))
```

We can now replace "y" with a sequence of function calls.

Each time we iterate, we need to process a new member of F; initially we are set up to get the first member. The following sequence, then, is needed:

```
L/CF(   L/OPEN/MEMBER(L/TOBJ(OPF),GET),
        z
        L/CLOSE/MEMBER(L/TOBJ(OPF)),
        L/WHICH/MEMBER(L/TOBJ(OPF),NEXT) )
```

The above is a compound function which will open the current member of F, do something to it (represented above by "z"), close it, and ask for the next member.

We want to replace "z" by a function call which tests the contents of A in the current member of F, and either does nothing or adds a member to G, copying the values of the current member of F. If "w" represents the action of adding a member to G and copying the values, then we can express this:

```
L/IF(L/EQUAL(L/TOBJ(M.A),LC/STRING('AB')),w)
```

A suitable way to express "add a member and copy values" is:

```
L/CF(L/OPEN/MEMBER(L/TOBJ(OPG),ADD),
        L/ASSIGN(L/TOBJ(GM.A),L/TOBJ(M.A)),
        L/ASSIGN(L/TOBJ(GM.B),L/TOBJ(M.B)),
        L/CLOSE/MEMBER(L/TOBJ(OPG))
```

This is similar enough to the previous example so that no explanation should be necessary.

Putting this all together, we get:

```
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),
 L/CF(  L/OPEN/MEMBER(L/TOBJ(OPF),GET),
        L/IF(L/EQUAL(L/TOBJ(A),LC/STRING('AB')),
                L/CF(   L/OPEN/MEMBER(L/TOBJ(OPG),ADD),
                        L/ASSIGN(L/TOBJ(GM.A),L/TOBJ(M.A)),
                        L/ASSIGN(L/TOBJ(GM.B),L/TOBJ(M.B)),
                        L/CLOSE/MEMBER/L/TOBJ(OPG)) ) )
        L/CLOSE/MEMBER(L/TOBJ(OPF)),
        L/WHICH/MEMBER(L/TOBJ(OPF),NEXT) ) )
```

To conclude the operation, we execute:

```
L/LISTOP/END(L/TOBJ(OPG))
L/LISTOP/END(L/TOBJ(OPF))
```

The result is a LIST G whose first member has value ('AB','CD'), and whose second member has value ('AB','IJ'). With a few variations on the above example, quite a few LIST operations can be performed.

4.11 Higher level functions

While these primitive i/functions are useful, we would not ordinarily expect users to operate in datalanguage at this low level. We want to make these primitives available to users so that they can handle the exceptional case, and so that they can construct their own high-level functions for atypical applications. Ordinarily, they ought to operate at least at the level of the following construction (which is legal in the real datalanguage currently implemented):

```
    FOR G.R,F.R WITH A EQ 'AB'
            G.R=F.R
            END
```

This relatively concise expression accomplishes the same result as the elaborate construction of i/functions given at the close of the preceding section. We could define i/functions very similar to the semantic functions used in the running software, and write the above request as:

```
    L/FOR(L/POBJ(G),R
            L/POBJ(F),R,L/WITH(L/EQUAL(L/TOBJ(A),
                                    LC/STRING('AB')))
```

The differences between the i/function call and the datalanguage request above it are principally syntactic.

In designing functions such as L/FOR and L/WITH, the central problems have to do with choosing the right restrictions. One cannot have all the generality available at the primitive level. Some important choices for these particular functions are: (1) handling multiple inputs and outputs, (2) when FORs are nested, how outer FORs restrict the options available to inner FORs, (3) generality of selection functions (may then in turn generate FORs?), (4) options with regard to where processing should start (are we updating, replacing or appending to the output list(s)?).

Finally, this problem is related to the more general problem of dealing with _sets_, which are a generalization of the idea of a collection of members in a LIST having common properties. FOR is only one of many operators that can input sets.

4.12 Conclusion

The present model, though embryonic, already contains enough primitives and data types to permit definition and generalized manipulation of hierarchical data structures. Common data management operations, such as retrieval by content and selective update can be expressed.

The use of this model in developing these primitives has resulted in precise, well-defined and internally consistent specifications for language elements and processing functions. Operating in the laboratory environment provided by the model seems to be a substantial benefit.

5. Further Work

In this section, we review what has been accomplished so far in the design and describe what work remains to be done before this design iteration of datalanguage is complete.

5.1 A Review

Most important, among our accomplishments, we feel that we have delineated the problems and presented the broad outlines of a solution to development of a language for the datacomputer system. Key elements of our approach are the primacy of data description in capturing all the aspects of the data, the separation of logical and physical characteristics of data description, the ability of users to define different views of the same data, the ability to associate functions with different uses of data items, an attempt to capture common aspects of data at every possible level, and the ability of users to communicate with the datacomputer in as high a level as their application permits.

5.2 Topics for Further Research

Although more work needs to be done in general to turn out a finished design for datalanguage, we can single out certain issues which in particular need further investigation.

So far, only hierarchal data structures (i.e. those that can be modeled by physical containment) have been developed to any extent. We also intend to investigate and provide other types of data structures. We are confident that our language framework does not make assumptions that would prohibit such additions.

Our current work on access regulation centers on the use of multiple descriptions for data. We need to do more work on both the technical and administrative aspects of access regulation. Problems of encrypting data for both transmission and storage will also be investigated.

Another issue requiring further research is the protocol requirement for process interaction with the datacomputer.

Separation of the description into independent modules needs further research. In particular, we need to look into work which has already been done on separate specifications of logical descriptions, physical descriptions, and mappings between the two.

5.3 Datalanguage Syntax

We have not yet proposed a syntax for the datalanguage we are developing. Certainly the most difficult parts of the problem have been the semantic and pragmatic issues. We are confident that various syntactic forms can be chosen and implemented without excessive difficulty. It may be best to develop different syntactic forms for the language for different types of users or even for the various subparts of the language itself. As discussed in section 2, the user syntax for the datacomputer is supposed to be at a low level. It should be easy for _programs_ to generate datalanguage requests in this syntax.

5.4 Further Work on the Datalanguage Model

The model provides an excellent foundation on which to build up a language with the facilities described in section 3. Much work is yet to be done.

For a while, emphasis will be on sets, high-level operators, language extension and data description.

We expect to model sets as a new datatype, whose value is ordinarily shared with other objects. Some further work on binding and sharing of values is needed to support this.

Sets can be regarded as a special case of generalized relations, which will come somewhat later.

High-level operators such as FOR will be constructed from the existing primitives, and will eventually be defined to have one effect but several possible expansions. The expansion will depend on the representation of the data and the presence of auxiliary structures.

Alternate expansions will be possible when the data description has been broken up into its various modules. This, also, requires some further research.

We feel that the language extension problem is much more easily attacked in the environment provided by the model datacomputer. In particular, we expect the laboratory environment to be helpful in evaluating the complex interactions and pervasive effects of operators in the language which extend the language.

Data description work in the near term will focus on the isolation of attributes, the representation of variable structure in description, the description of descriptions and the development of a sufficient set of builtin data types.

Later, we expect to model the semantics of pointers as a datatype, when the representation of the pointer and the semantics of the address space into which it points are specified in the description of the pointer.

A large number of lower-level issues will be attacked, including some of the problems discovered in the modeling to date. Some of these are pointed out in the discussions in section 4.

5.5 Applications Support

The datalanguage we are designing is intended to provide services to sub-systems solving a broad class of problems related to data management. Examples of such sub-systems are: report generators, online query systems for non-programmers, document-handling systems, transaction processing systems, real-time data collection systems, and library and bibliographic systems. There are many more.

The idea is that such systems will run on other machines, reference or store data at the datacomputer, and make heavy use of datalanguage. Such a system would not be written entirely in datalanguage, but a large component of its function would be expressed in datalanguage requests; some controlling module would build the requests and perform the non-datalanguage functions.

While we have experience with such applications in other environments, and we talk to potential users, it will require some work to determine that our language is actually adequate for them. That is, we are not attacking directly the problem of building a human-oriented online query system; we are trying to provide the tools which will make it easy to build one. There is a definite need to analyze whether the tools are likely to be good enough. Of course, the ultimate test will be in actual use, but we want to filter out as many problems as we can before implementation.

An important component of supporting applications is that the using programs will frequently be written in high-level languages such as FORTRAN, COBOL or PL/1. We will want to investigate the ability of datalanguage to support such users, while the design is taking shape.

5.6 Future Plans

This paper has laid the foundations for a new design of datalanguage. Section 3 provides an outline for a datalanguage design, which will be filled in during the coming months. Following the issue of a detailed specification, we anticipate extensive review, revisions, and

incorporation into the implementation plans. Implementation will occur in stages, compatible with the established plans for development of datacomputer service and data management capabilities.